Institut für Parallele und Verteilte Systeme
Abteilung Verteilte Systeme
Universität Stuttgart
Universitätsstraße 38
D70569 Stuttgart

Studienarbeit Nr. 2113

Churn in dynamischen Publish/Subscribe Systemen

Beate Ottenwälder

Studiengang:	Informatik
Prüfer:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Betreuer:	Dr. Boris Koldehofe
begonnen am:	11. Juni 2007
beendet am:	11. Dezember 2007
CR-Klassifikation:	C.2.1, C.2.4, D.4.4

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

APT
: Acess-Point-Lookup-Table

FEL
: Future Event List

GUI
: Grafische Benutzerschnittstelle

NONCE
: Einmal verwendete Nummer

P2P
: Peer-to-Peer

pub/sub
: Publish/Subscribe

RPC
: Entfernter Prozeduraufruf

RPF
: Reverse Path Forwarding

Kapitel 1

Einleitung

Moderne Verteilte Anwendungen, wie RSS-Feeds [4] und Online-Börsenhandel [5], werden immer häufiger in großen Verteilten Systemen eingesetzt. Um in solchen großen und flexiblen Systemen die Kommunikation skalierbar zwischen allen Teilnehmern zu gestalten muss ein Kommunikationsparadigma verwendet werden, welches dieser Problematik gerecht wird. *Publish/Subscribe (pub/sub)* ist ein many-to-many Kommunikationsparadigma, welches diese Anforderungen erfüllt. Durch das erhöhte Interesse an diesem Paradigma ist es sinnvoll, die entsprechenden Systeme auf ihre Leistungsfähigkeit zu untersuchen.

In einem Topic-basierten pub/sub-System schreiben sich *Subscriber* zu einem Topic bei einem Ereignis-Dienst ein, um alle von *Publishern* zu diesem Topic erzeugten Ereignisse zu erhalten. Bei pub/sub-Systemen mit einer dezentralen Architektur wird keine zentrale Instanz benötigt. Dennoch müssen Publisher und Subscriber bei jeglicher Systemgröße interagieren können. Wünschenswert wäre, wenn jeder Teilnehmer des Systems lokal und autonom arbeiten, aber dennoch am globalen System teilhaben könnte. Bei *Peer-to-Peer (P2P)* Systemen ist eine solche skalierbare, autonome und dezentrale Organisationsmöglichkeit schon vorzufinden. Deshalb ermöglicht es diese Technik ein dezentrales pub/sub-System aufzubauen. Es existieren schon diverse Systeme, die diese Idee aufgegriffen haben, beispielsweise Scribe [6]. Häufig werden dabei Teilnehmer, welche das selbe Interesse an einem Topic besitzen, in Gebieten, also der selben Nachbarschaft, zusammengefasst.

Churn gibt die Häufigkeit, mit der Prozesse einem Verteilten System beitreten und es wieder verlassen, an. Im Falle der pub/sub-Systeme wird Churn durch Subscriptions und Unsubscriptions von Prozessen bestimmt. Die Motivation, Churn in pub/sub-Systemen anders zu betrachten als in anderen Verteilten Systemen, ist durch die Annahme gegeben, dass in einem interessanten Topic die Rate der Subscribtions und Unsubscribtions, also dem

Churn, größer ist als in einem uninteressanten. Währendessen geht man in den meisten anderen Systemen davon aus, dass das Interesse über alle Teilnehmer und Gebiete gleichverteilt ist.

In existierenden pub/sub-Systemen [5, 6, 7] wurden dabei Untersuchungen bezüglich der Thematik dieser Arbeit fast immer auf die Belastbarkeit bezüglich Churn beschränkt. Insbesondere wird in diesen Systemen eine Lastverteilung angestrebt. Last ist hierbei durch die Aufgaben der Teilnehmer, Nachrichten weiterzuleiten und die Struktur aufrecht zu halten, vorhanden. Bei einer fairen Lastverteilung soll allerdings anhand des Nutzens durch Ereignisse Arbeit verteilt werden. Churn verursacht durch die Suche nach einem Einstiegspunkt in ein Gebiet eines Topic Arbeit bei Teilnehmern, welche die Suchanfrage weiterleiten müssen und nicht zwangsläufig im Zieltopic liegen. Dieses Verhalten kann die Arbeit unfair verteilen. Häufig werden die selben Teilnehmer, die nicht an dem durch Churn belasteten Topic Interesse haben, durch diese Suchanfragen belastet. Dieses Verhalten ist in Systemen mit gleichverteiltem Churn kein Problem, denn dort wird die Arbeit ebenfalls über die Teilnehmer gleichverteilt. Dies wäre in diesem Fall fair. In Systemen, in denen der Churn unterschiedlich auf die Topics verteilt ist, kommt es in vielen Szenarien dazu, dass Teilnehmer in Gebieten mit wenig Churn die Arbeit dafür tragen müssen Suchanfragen, in Gebiete mit hohem Churn, weiterzuleiten. Im Folgenden wird auf die Fairness im Zusammenhang von Arbeit durch Churn eingegangen werden. Fair ist ein System, wenn Knoten innerhalb eines Topic mit wenig Churn nicht mehr Arbeit durch Topics mit viel Churn erhalten. Um dies zu gewährleisten wurde untersucht, durch welche Mechanismen die Mehrarbeit in anderen Topics entsteht und wie diese Arbeit dann gerechter verteilt werden kann.

In dieser Arbeit wird ein P2P-basiertes pub/sub-System vorgestellt und evaluiert, welches die Möglichkeit bietet, Arbeit durch Churn fairer zu verteilen. Das System baut auf einem strukturierten, Skip-Listen- sowie Gossip-basierten P2P-Overlay namens *GosSkip* [2] auf. Die Teilnehmer, welche an verschiedensten Topics Interesse zeigen, werden dabei pro Topic gruppiert und so innerhalb der Struktur platziert.

An dem entwickelten System wurden Untersuchungen bezüglich des Verhaltens unter Churn angestellt. Dabei wurde zum einen ein Ansatz, welcher mit einer lokalen Sicht und zum anderen ein Ansatz, welcher mit einer globalen Sicht arbeitet, entwickelt. Hierbei wurde die Eigenschaft von GosSkip, lookup-Anfragen anhand von unterschiedlich großen Gebieten fair zu verteilen, ausgenutzt. Mit diesen Methoden wurde erreicht, dass die Arbeit mehr auf Teilnehmer, in Gebieten mit hohem Churn, verlagert wurde und deshalb sich das Systemverhalten fairer gestaltet. Die Arbeit so fair zu verteilen, dass es dem Churn gerecht wird, ist nicht komplett möglich, kann aber, wie hier

gezeigt wird, angenähert werden. In einer parallelen Studienarbeit [8] wurde dieses System bezüglich fairer Arbeitsteilung untersucht.

Struktur. Diese Arbeit ist folgendermaßen aufgebaut: Zuerst werden die Grundlagen zur Thematik dieser Arbeit und der dabei zur Anwendung gebrachten Techniken erläutert. Im Anschluss daran werden verwandte Arbeiten vorgestellt. Der Churn als Begrifflichkeit, sowie als Problematik wird im darauffolgen Kapitel besprochen. Als nächstes wird die Architektur - das pub/sub-System - konzeptionell beschrieben. Die Evaluierung bezüglich des Systems und der Methoden Churn zu beheben werden danach behandelt. Als letztes werden Schlussfolgerungen gezogen und Ausblicke gegeben.

Kapitel 2

Grundlagen

2.1 Publish/Subscribe

Publish/Subscribe (pub/sub) ist ein many-to-many Kommunikationsparadigma, das den Bedürfnissen heutiger großer Systeme, wie dem Internet, genügt. Dieses Paradigma erfreut sich immer größerer Beliebtheit und kommt in vielen verteilten Anwendungen zum Einsatz. Beispiele dafür sind RSS-Feeds [4] und Online-Börsen-Handel [5]. Durch das erhöhte Interesse an diesem Paradigma ist es sinnvoll, die entsprechenden Systeme auf ihre Leistungsfähigkeit zu untersuchen. Diese ist gegebenenfalls zu optimieren.

Publish/Subscribe ist neben Nachrichtenaustausch (Message passing), Remote Procedure Call (RPC) und Nachrichten Warteschlangen (Message Queing) eines der vielen Kommunikationsparadigmen. Bei pub/sub werden Informationen von vielen sogenannten *Publisher*-Prozessen generiert und an viele *Subscriber*-Prozesse asynchron, in einem flexiblen und lose gekoppelten System verteilt. Die generierten Informationen werden *Ereignisse* und die Nachrichtenauslieferung *Benachrichtigung* genannt. Ein Teilnehmer dieses many-to-many Kommunikationssystems ist nicht auf eine Rolle als Subscriber oder Publisher festgelgt. Er kann beides zugleich sein.

Der größte Unterschied zu anderen Kommunikationsparadigmen liegt in der Entkoppelung von *Zeit*, *Raum* und *Synchronisation*. Dies bedeutet, dass die Kommunikationspartner nicht gleichzeitig aktiv an der Interaktion beteiligt sein müssen (Zeit), sich nicht kennen müssen, geschweige denn wissen müssen wieviele Entitäten an der Interaktion beteiligt sind (Raum) und ihre Aktionen asynchron ausführen können (Synchronisation). Diese Entkoppelungseigenschaft trägt zur Skalierbarkeit des Systems bei. Sowohl Publisher als auch Subscriber können unabhängig voneinander arbeiten. Dies ist in Anbetracht der Größe und der Unvorhersehbarkeit des Internets, sowie anderer

riesiger Systeme, zwingend notwendig.

Das prinzipielle Systemmodell bei pub/sub beinhaltet neben den Publishern und Subscribern einen *Ereignis-Dienst*. Jeder Subscriber-Prozess schreibt sich bei einem Ereignis-Dienst für Ereignisse, die ihn interessieren, ein. Diesen Vorgang bezeichnet man als *Subscription*. Ein Subscriber wird dann benachrichtigt, wenn ein Ereignis, welches zu seiner Einschreibung passend ist, von einem Publisher-Prozess generiert wurde. Wenn ein Prozess kein Interesse mehr an bestimmten Ereignissen hat, trägt dieser sich mit einer *Unsubscribe*-Möglichkeit beim Ereignis-Dienst aus.

Die Subscription kann sowohl mit *Topic-basierten*, *Content-basierten* als auch *Type-basierten* Schemata geschehen. Hierbei wird angegeben, an welchen Ereignissen ein Prozess Interesse hat, bzw. an welchen dieser kein Interesse hat. Bei Topic-basierten Schemata wird ein Schlüsselwort angegeben und der Subscriber schreibt sich damit zu einem bestimmten Topic ein. Dies ist vergleichbar mit dem Beitritt einer Gruppe bei Gruppenkommunikation. Content-basiertes pub/sub dagegen basiert nicht auf externen Beschreibungsmitteln wie einem Topic, sondern arbeitet mit den Eigenschaften der Ereignisse selbst. Welche davon den Subscriber interessieren, wird beim Subscription-Vorgang mit Hilfe von Filtern angegeben. Dies basiert meist auf Vergleichen von name-value Paaren. Beispielsweise könnte man stringbasiert durch Angabe von $Timestamp < \text{"}14:00\text{"}$ nur über alle Ereignisse vor 14 Uhr informiert werden. Bei Type-basierten Subscriptionmethoden wird der Ereignis-Typ als Schema, welches über Ereignisse bestimmt, die ein Subscriber erhalten soll, herangezogen.

Wenn man verschiedene Architekturen anschaut, so kann man zwei Kategorien unterscheiden. Zum einen *zentralisierte Architekturen* [9, 10], bei denen Nachrichten Broker, bzw. eine Menge an Brokern, zentrale Entitäten darstellen, das Ereignis des Publishers speichern und auf Anfrage an den Subscriber weiterleiten. Zum anderen *dezentrale Architekturen* [6, 7], bei denen keine Broker vorhanden sind, sondern direkt via IP Multicast oder anderen Mechanismen ein Ereignis zum Subscriber gelangt. In diesem Fall haben sowohl Publisher-Prozesse als auch Subscriber-Prozesse Ereignisse zu speichern und weiterzuleiten.

Diskussionen über zentrale und dezentrale Architekturen kann man auch bei P2P-Systemen finden. Diese dezentralen Systeme grenzen sich gegen zentralisierte Systeme, wie beim Client-Server Prinzip zu finden, ab. Nicht nur deshalb sind P2P-Systeme attraktiv für pub/sub-Lösungen. Da P2P-Systeme dezentral sind, gut mit der Systemgröße skalieren, autonom Entscheidungen innerhalb des Systems treffen und dennoch eine Verbundenheit der Knoten herrscht, kann man diese Technologie als Grundlage für ein dezentrales pub/sub-System verwenden. Beispielsweise kann man als Subscription zu Topics

Knoten erzeugen und dem P2P-System beitreten lassen oder einfach einem Knoten im P2P-System eine Anfrage stellen um andere Teilnehmer, die an den selben Topics Interesse haben, im System zu finden. Aus diesem Grund wird im nächsten Kapitel näher auf P2P-Technologien im Allgemeinen eingegangen.

Zeitentkoppelung erweist sich allerdings in dezentralen Architekturen als schwer realisierbar. In einem System, in dem Subscriber-Prozesse offline gehen und zu einem beliebigen Zeitpunkt in der Zukunft oder nie wieder erscheinen können, werden diejenigen bestraft, welche online bleiben und für andere Subscriber Ereignisse lange vorhalten müssen. Ein Prozess, welcher als einziger ein bestimmtes Ereignis vorrätig hat, könnte abstürzen und das Ereignis wäre für nachfolgende Prozesse möglicherweise verloren. Man kann einem Subscriber, welcher während er eingeschrieben ist offline geht, nicht garantieren, dass ihn jedes Ereignis erreicht. Es ist eine sinnvolle Einschränkung, bei dezentralen Systemen Zeitentkoppelung außer Acht zu lassen. Laut Tanenbaum et al. [11] haben die meisten pub/sub-Systeme eine zeitliche Koppelung.

In dieser Arbeit wird eine dezentrale Architektur verwendet, um ein Topic basiertes pub/sub System zu verwirklichen. [11, 12]

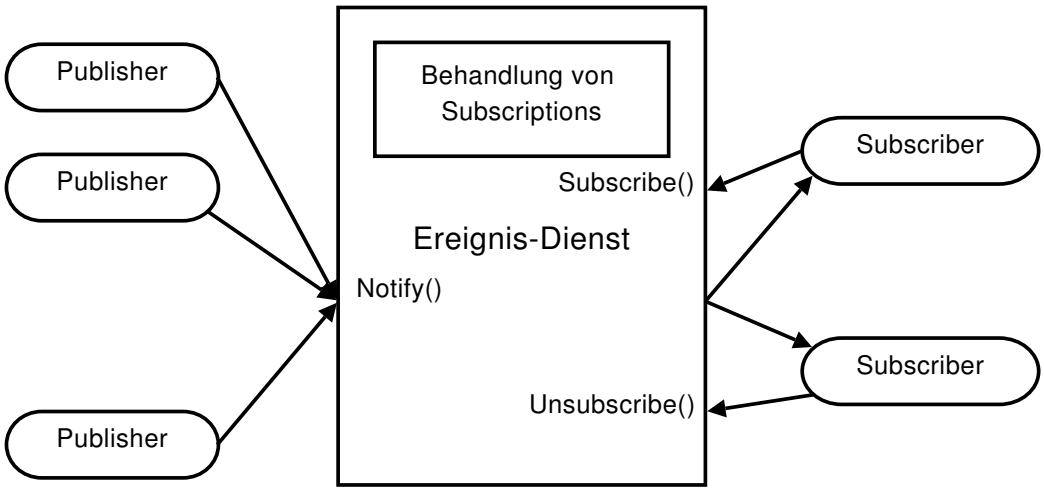

Abbildung 2.1: Publish/Subscribe

2.2 Peer-to-Peer

Ein *Peer-to-Peer* System ist ein Verteiltes System, in dem jeder Knoten gleiche Aufgaben übernimmt. Im Gegensatz zu Client-Server Systemen (Abbil-

dung 2.2) spricht man bei P2P-Systemen an Stelle von Knoten oft auch von Peers oder Servants (einem Kunstwort aus Server und Client). Dies begründet sich darin, dass Peers bzw. Servants sowohl Client- als auch Server-Aufgaben übernehmen. Aus dieser Rollensymmetrie ziehen diese Systeme Vorteile in Hinsicht auf *Skalierbarkeit*, *Fehlertoleranz* und der Vermeidbarkeit einer *"einzelnen Fehlerstelle"*. Diese Vorteile werden allerdings mit einer höheren Komplexität bezahlt. Für gewöhnlich halten sich Peers in *Nachbarschaftstabellen* oder *Nachbarschaftslisten* Informationen über eine Teilmenge der am System partizipierenden Peers. Die Peers in dieser Tabelle werden dementsprechend Nachbarn genannt. Um die Architektur des Systems zu beschreiben fällt häufig der Begriff Overlay Netz. Dabei handelt es sich um eigenständige Netzwerk-Topologien, die auf eine bestehende Netzwerk-Topologie, auch Underlay genannt (z.B. dem Internet), aufgelegt sind.

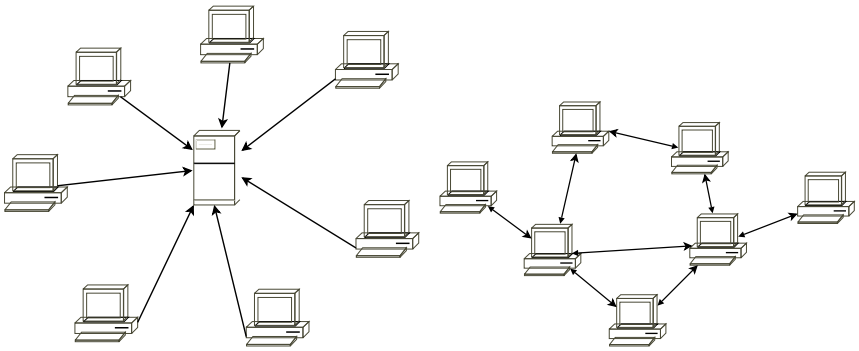

Abbildung 2.2: Client-Server gegenüber Peer-to-Peer

Auch wenn die wenigsten der existierenden P2P-Systeme die folgenden Eigenschaften komplett erfüllen, so charakterisiert man diese zumeist folgendermaßen:

Rollensymmetrie: Peers übernehmen Client- und Serverfunktionalität.

Dezentralisierung: Ein Peer hat weder die globale Sicht, noch trägt er die Gesamtlast des Systems. Das System kommt ohne einen zentralen Koordinator aus.

Selbstorganisation: Lokale Interaktionen führen zu einem komplexen, globalen Verhalten.

Autonomie: Jeder Peer kann autonom Entscheidungen treffen.

Zuverlässigkeit: Peers und Netzwerkverbindungen sind grundsätzlich Unzuverlässig. Mit entsprechenden Strategien wird dies kompensiert um beispielsweise Ausfälle zu verkraften.

Verfügbarkeit: Alle Informationen in diesem dezentralen System müssen jederzeit, auch beim Ausfall einzelner Knoten, verfügbar sein.

Der Grad an Strukturiertheit variiert stark von System zu System. Man unterscheidet grob zwischen strukturierten und unstrukturierten P2P-Systemen. Auf beiden Systemarten aufbauend sind schon pub/sub-Systeme entwickelt worden. Bei unstrukturierten P2P-Systemen sind die Knoten ohne eine vorgegebene Topologie verbunden und kein Peer hat Informationen über die Ressourcen anderer Peers. Dies führt bei Suchanfragen dazu, dass, wie in Gnutella, Querys durchgeführt werden müssen. Man versieht hierbei die Suchanfrage mit einer Lebensdauer (TTL) und versendet diese an eine Teilmenge der Nachbarn. Diese TTL wird nach Empfangen dekrementiert und, falls sie noch nicht 0 beträgt, weitergesendet. Zu dieser Klasse der Systeme gehören Gnutella [13] und BitTorrent [14]. Bei strukturierten P2P-Systemen ist meist in der Anordnung der Peers eine Struktur, wie einem Ring oder einem Torus, zu erkennen. Bei den entsprechenden Peers werden Informationen über die Ressourcen anderer Peers gehalten. Die Suche, hier auch *lookup* genannt, kann dann gezielt anhand spezifischer Informationen, wie einem Hashwert, durchgeführt werden. Sehr häufig verwendet man hier das Prinzip eine Nachbarschaftstabelle mit einer Höhe zu versehen. Je weiter ein Nachbar in der Topologie entfernt ist, desto höher wird er in der Liste eingetragen. Vertreter dieser Klasse sind Chord [15], Can [16] und GosSkip [2].

In diesem Zusammenhang wird sehr häufig von verteilten Hash-Tabellen (DHT) und konsistentem Hashing gesprochen. Dabei werden Informationen über Ressourcen und Informationen über Peers durch Hashing in einen Schlüsselraum abgebildet. Einem Peer wird damit die Verantwortung für einen bestimmten Teilraum der Schlüssel übertragen. Anhand dieser Schlüssel und entsprechenden Nachbarschaftstabellen kann die weiter oben erwähnte und für strukturierte P2P-Systeme typische gezielte verteilte Suche durchgeführt werden. In diesen Nachbarschaftlisten werden meistens nur genug Informationen gehalten, um in durchschnittlich $\log(N)$ hops, bei N Knoten im System, sein Ziel zu erreichen. So werden auch nicht alle Knoten, welche an einem P2P-System teilnehmen, in einer Nachbarschaftstabelle gespeichert, denn dies skaliert nicht mit der Größe der einzelnen Systeme. Bei Chord werden beispielsweise solche DHT 's verwendet. Konsistentes Hashing weist Elementen mit hoher Wahrscheinlichkeit einen Schlüssel so zu, dass alle Elemente gleichverteilt über den Schlüsselraum liegen. Jederzeit soll bei einer Anzahl von N Knoten und einer maximalen Anzahl an K Schlüsseln jeder Knoten für $\frac{(1+\varepsilon)*K}{N}$ Schlüssel verantwortlich sein.

Ein Problem von P2P-Systemen ist das Auffinden eines Knotens, der schon im System ist. Man spricht hierbei von *Bootstrapping*. Dies ist elementar, da

man sonst dem System nicht beitreten kann. Dabei gibt es eine weite Spanne von Möglichkeiten Bootstrapping durchzuführen. Beim ursprünglichen Bittorrent beispielsweise existiert dazu eine zentrale Instanz, der Tracker, und sogenannte ".torrent"-Dateien um diesen ausfindig zu machen.

Ein P2P-System ist nicht nur zum Filesharing, durch welches diese Art der Technologie berühmt geworden ist, einsetzbar. Man kann, wie in dieser Arbeit gezeigt wird, die Fähigkeiten eines solchen Systems als Grundlage für ein pub/sub-System verwenden. Eine skalierbare und fehlertolerante Struktur, über die man kommunizieren kann, entspricht den Anforderungen eines pub/sub-Systems und kann deshalb als fundierte Grundlage dazu verwendet werden Subscriber eines Topics ausfindig zu machen oder zu gruppieren. Dezentralisierung, autonomes Handeln und Selbstorganisation der Peers erfüllen Vorrausetzungen, welche für ein dezentrales pub/sub-System notwendig sind. [15, 17, 18]

2.3 Skip Listen

In dieser Arbeit werden *Perfekte Skip Listen* von Bedeutung sein. Die Prinzipien von Listen und Bäumen werden in Verteilten Systemen häufig aufgegriffen. Man spricht dann von Multicastbäumen oder von Ringen, wie beim Chord-Ring. Ein Teilnehmer an einem Verteilten System wird oft als Knoten bezeichnet. Dieser hat Verbindungen zu anderen Knoten. Diese Verbindungen kann man, wenn man sie auf Listen bezieht, als Pointer zum nächsten Element sehen. Um nun die, für die Arbeit verwendeten grundlegenden Prinzipien zu erläutern, werden Architektur und Eigenschaften, wie Suchaufwand in $O(\log(N))$, von *Skip Listen* vorgestellt.

Bei *Skip Listen* handelt es sich um mehrfach verkettete Listen, die anstelle von balancierten Bäumen verwendet werden können. Ein jedes Element ist anhand eines Schlüssels in der Liste einsortiert. Diesen Element-Schlüssel 2-Tupeln werden zufällig Höhen $\mathrm{i} \in \mathbb{N}$ zugewiesen. Verbunden sind sie dann immer mit den nächsten Nachbarn auf Ebene $\mathrm{j} \in \{\mathrm{k} \mid \mathrm{k} \in \mathbb{N} \wedge k \leq i\}$. Anstatt einer Liste kann man alternativ auch einen Ring auf diese Art aufbauen.

Eine solche Liste besitzt eine maximale Höhe h. Diese maximale Höhe kann durch $O(\log(N))$, wobei N die Anzahl der Elemente ist, beschränkt werden. Ein sogenannter Listenkopf ist der Startpunkt für die Suche, das Einfügen und das Löschen. Er besitzt h Verbindungen zu Nachfolgern. Elemente innerhalb der Liste, die keinen Nachfolger auf Ebene j besitzen, zeigen auf NULL.

Eine Suche erfolgt hier, indem man am Listenkopf die Suche auf der obersten Ebene startet. Man folgt den Verbindungen auf oberster Ebene (Höhe

h) solange die nächsten Schlüssel der Elemente in der Sortierung vor dem gesuchten Schlüssel liegen (kleiner bei einer aufsteigenden Sortierung von Natürlichen Zahlen). Kommt man zum letzten Element, welches vor dem gesuchten Schlüssel in der Sortierung ist, muss man mit der Suche eine Ebene weiter unten fortfahren. Dieser Iterationsschritt wiederholt sich bis man auf Ebene 0 nicht mehr weiter kommt, dann ist man am Ziel. Die Kosten sind hierbei im average case: $O(\log(N))$, wobei N die Anzahl der Elemente ist.

In Abbildung 2.3 ist exemplarisch die Suche nach einem Element mit Schlüssel x $\in \{k \mid k \in \mathbb{N} \wedge 12 \leq k \leq 20\}$ dargestellt. Nimmt man für x den Wert 18, so verläuft die Suche wie folgt. Am Listenkopf beginnt die Suche. Es wird festgestellt $18 < 21$, also wird eine Ebene weiter unten gesucht. Als nächstes wird $9 < 18$ ermittelt und beim 2-Tupel mit Schlüssel 9 weitergesucht. Nachdem man zwei mal den bool'schen Audruck $18 < 21$ als wahr auswertet, kann man schlussendlich auf Ebene 0 zum 2-Tupel mit Schlüssel 12 weitergehen. Dort wird festgehalten, dass die Suche nicht weitergehen kann und entweder kann nun hier ein Element eingefügt werden oder man weiß, dass das Element mit dem gesuchten Schlüssel 18 nicht existiert.

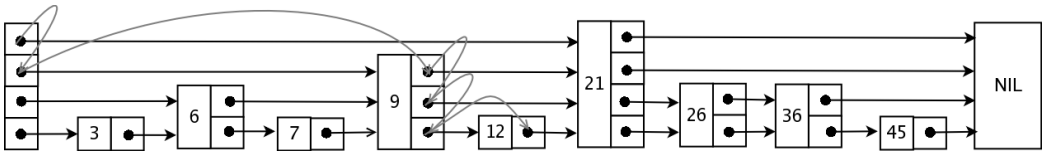

Abbildung 2.3: Suche nach dem Eintrag mit Schlüssel 18 bei einer randomisierte Skipliste [1]

Eine *Perfekte Skip Liste* springt mit einer Verbindung auf Ebene j exakt zum 2^j-ten Nachfolgerelement. Es ergibt sich damit, dass jedes 2^0-te Element eine Verbindung zum nächsten, jedes 2^1-te Element eine Verbindung zum übernächsten besitzt, wobei sich diese Reihe beliebig erweitern lässt. 50% der Elemente haben also die Höhe 1, 25% der Elemente haben die Höhe 2 und so weiter. Anhand dieser Verteilung, welche die Anzahl der Elemente, die eine bestimmte Höhe haben, immer halbiert, kann man die maximale Höhe auf log(N) abschätzen. Sei N die Anzahl der Elemente, so lässt sich die Anzahl der Verbindungen folglich durch

$$|Listenkopf| + 1 + \sum_{l=0}^{log(N)-1} \frac{N}{2^l}$$

berechnen. [2, 1]

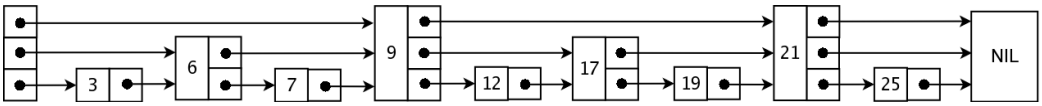

Abbildung 2.4: Perfekte Skipliste [1]

2.4 Gossip Algorithmen

Gossip-basierte, oder epidemisch genannte Algorithmen erlauben es Informationen auf gleiche Weise im System zu verbreiten, wie sich Gerüchte oder Viren unter Menschen verbreiten. Wenn z.B in einer Abteilung der Universität Mitarbeiter A Informationen über ein Meeting an Mitarbeiter B, Mitarbeiter C und Mitarbeiter D ausplaudert, so können Mitarbeiter B, Mitarbeiter C und Mitarbeiter D diese Informationen zusätzlich weiter erzählen. Wenn dann Mitarbeiter B die Information an Mitarbeiter E weiterleitet, so kann Mitarbeiter E fortan die Nachricht im Kollegium verbreiten. So wird die Informationsverbreitung zum Selbstläufer. Sollte nun Mitarbeiter B die Information nicht richtig verstanden haben oder ab sofort Lügen verbreitet, so ist das nicht schlimm, denn mit hoher Wahrscheinlichkeit trifft er irgendwann auf einen Mitarbeiter, der schon von jemand anderem informiert worden ist und erfährt so die korrekten Daten des Meetings. Macht Mitarbeiter B früher Feierabend als die anderen und erzählt deswegen niemandem von dem Meeting, so wissen zumindest Mitarbeit A, Mitarbeiter C und Mitarbeiter D davon und können es immer noch weiterpropagieren.

Wie das obige Beispiel verdeutlicht, haben epidemische Algorithmen große Vorteile in Hinblick auf *Robustheit*, *Zuverlässigkeit*, *Skalierbarkeit* und *Fehlertoleranz*. Als schwierig betrachtet werden kann, dass möglicherweise Prozesse Informationen bekommen und weiterleiten, die sie nicht interessieren und dass ein ständiger Nachrichtenaustausch im Hintergrund stattfindet.

Für gewöhnlich haben gossip-basierte Algorithmen eine festgelegte Puffergröße b, die bestimmt, wieviele Informationen ein Prozess zwischenspeichern muss und einen Parameter t der bestimmt, wie häufig ein Prozess diese Information an eine zufällige Teilmenge M ($\subseteq$ G, sei G die Menge aller Prozesse im System) von anderen Prozessen weiterträgt. Die Parameter t, b und m=$|M|$ können statisch sein, was sich jedoch mit steigender Größe des Systems n=$|G|$ negativ auf die Zuverlässigkeit auswirkt. Die Parameter können sich aber auch, z.B. logarithmisch $log(n)$, während des laufenden Betriebs an n anpassen.

Gossip-basierte Ansätze können für dynamische pub/sub-Systeme eine wertvolle Ergänzung sein, da sie Informationen über die Teilnehmer des Systems verteilen können. [19]

Kapitel 3

Verwandte Arbeiten

3.1 GosSkip

Wenn man *Perfekte Skip Listen* auf ein *gossip-basiertes*, *strukturiertes* P2P-System abbildet erhält man GosSkip. Die angeordneten Elemente, z.B. Dateien, werden hierbei als Peers bezeichnet und die Schlüssel sind Namen, die den Elementen, z.B. Dateinamen, zugeordnet sind. Die Menge der Namen muss unter einer Ordnungsrelation eine Vollständige Halbordnung bilden. Im Gegensatz zu anderen Systemen werden hier nicht einzelne Recheneinheiten, die als physikalische Knoten in der Arbeit bezeichnet werden, sondern einzelne Elemente, in dieser Arbeit auch (virtuelle) Knoten genannt, als Peers angeordnet.

Die grundlegende Overlay-Topologie stellt einen Ring dar - den GosSkip-Ring. Dieser Ring ist auf Ebene 0 eine Skip-Liste mit Höhe 0. Ein jeder Knoten in diesem System kennt Knoten, die rechts und Knoten, die links von ihm liegen. Diese werden in entsprechenden Nachbarschaftslisten gehalten. Es existieren Verbindungen zu Nachbarn auf gewissen Ebenen i $\in \mathbb{N}$, wobei die maximale Höhe der Nachbarschaftslisten durch $\lfloor \log(N) \rfloor$, sei N die Anzahl der Knoten, beschränkt ist. Die Metrik, welche ab sofort Aussagen über Nähe trifft, beschreibt die Abständen in hops auf dem grundlegenden GosSkip-Ring der Ebene 0.

Auf jeder Ebene wird periodisch eine *Gossip Nachricht* versendet. Das Versenden findet allerdings nur in eine der beiden Richtungen statt, abhängig davon auf welcher Ebene die Nachricht versendet wird. Nach rechts wird auf geraden Ebenen gesendet, nach links auf ungeraden. In dieser Nachricht wird eine Menge M an Tripeln von Nachbar zu Nachbar getragen. Diese Tripel halten Informationen über eine Identifikation (id, wie in der vorliegenden Implementierung beispielsweise die ip-Adresse), einen Namen eines Datenele-

ments, sowie einen Zähler, der bei jedem hop inkrementiert wird. Sobald ein Knoten eine solche Nachricht erhält, speichert er sich die Menge zwischen, fügt ein Tripel mit Informationen über sich selbst hinzu, um im nächsten Gossip Takt diese Menge weiter zu senden.

Die Skiplänge k $\in \mathbb{N}$ bestimmt, wie viele Verbindungen durch eine Verbindung auf Ebene i auf dem Ring, der auf Ebene i-1 verbundenen Knoten, übersprungen wird. Verbindungen auf Ebenen > 0 werden nach und nach während des laufenden Betriebs des Systems aufgebaut. Sobald ein Knoten auf Ebene i Gossip-Nachrichten mit counter=0 erhält weiß er, dass dies sein direkter linker bzw. rechter Nachbar auf dieser Ebene ist. Ebenso weiß er, sobald er eine Nachricht mit counter=k-1 erhält, dass dies sein Nachbar auf Ebene i+1 ist. Demzufolge kann man hier auch diesen Eintrag aus M entfernen, da niemand der darauffolgenden Knoten sich noch für den Eintrag interessiert.

In Abbildung 3.1 ist der Prozess für k=2 dargestellt. Wenn also k=2 ist und sei N die Anzahl der Knoten im GosSkip-Ring, so hat ein Knoten 2*$\lfloor \log(N) \rfloor$ Nachbarn. In diesem Fall, kann er mit den Verbindungen auf Ebene 0,1 ... $\log(N)$ hier direkt zu seinem entsprechenden, rechten oder linken 2^0-ten, 2^1-ten ... $2^{\log(N)}$-ten Nachbarn routen.

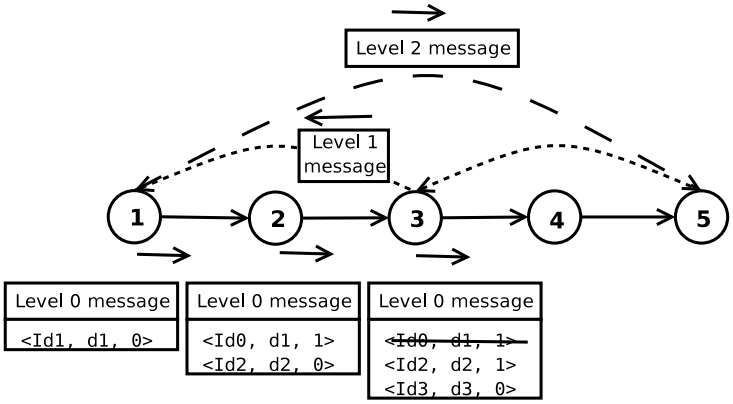

Abbildung 3.1: Periodische Gossip-Nachrichten auf allen Ebenen [2]

Die verteilte Suche nach einem Namen kann dann, entsprechend der Suche in einer *Skip-Liste*, implementiert werden. Eine lookup-Nachricht wird auf der obersten Ebene solange weitergesendet, bis der nächste Name eines Knotens in der Ordnung nach dem gesuchten Namen zu finden ist. So geht man von Ebene zu Ebene nach unten, bis man auf unterster Ebene das entsprechende Element gefunden hat.

Um dem System beizutreten, muss ein Knoten nur einen einzigen anderen Knoten im System kennen und von dort eine verteilte Suche nach seiner Po-

sition im Ring starten. Sobald die passende Stelle gefunden wird, fügt man den Knoten dort ein. Dieser Knoten kennt zu Anfang nur seine Nachbarn auf Ebene 0. Das ganze P2P-System konvergiert allerdings durch *Gossip Nachrichten* zu einer Struktur ähnlich einer *Perfekten Skipliste*. Knoten gehen, indem sie einfach aufhören periodische Gossip Nachrichten zu senden.

Der einfachste Ansatz hier ein Topic-basiertes pub/sub aufzubauen wäre es, einen Teilnehmer als virtuellen Knoten zu jedem Topic, welches ihn interessiert, in den GosSkip-Ring einzufügen. Dabei ist es sinnvoll Subscriber mit ähnlichem oder gleichem Interesse in eine Nachbarschaft zu bringen. So wird vermieden unnötigerweise Knoten mit Arbeit beim Publishen zu belasten, wenn sie nicht wirklich am Topic interessiert sind. Deshalb könnte die Sortierung der Namen hierbei längenlexikographisch anhand des Schlüsselworts geschehen. Publishing wäre mit Hilfe des in [2] vorgestellten Spreading Algorithmus effizient möglich.

Churn ist eine Bezeichnung für die Rate der Subscriptions und Unsubscriptions. Durch Churn werden bei GosSkip Knoten außerhalb des eigentlichen Topics durch Arbeit belastet. Dies geschieht zum einen, da lookup-Anfragen für die Position des Subscriber durchgeführt werden müssen und zum anderen, da lange Verbindungen durch Unsubscriptions kurzzeitig veraltet und dadurch unbrauchbar sind. Wie gezeigt werden konnte, ist das Verhalten dieses pub/sub-Systems unter Churn durch Subscriptions gut. Jedoch ist das Verhalten unter Churn durch Subscriptions als auch Unsubscriptions schlecht, da lange Verbindungen in das Topic verloren gehen und lookup Anfragen dadurch sehr viele hops benötigen, respektive mehrmals wiederholt werden müssen. [2, 20]

3.2 Scribe

Scribe ist ein pub/sub-System das, genauso wie das hier vorgestellte System, auf P2P aufbaut. Allerdings baut es nicht auf GosSkip, sondern auf dem DHT basierten Pastry [21] auf. Über Pastry wird eine Multicastbaum-Struktur für jedes Topic aufgebaut, um mit dessen Hilfe publishen zu können. Der Multicastbaum könnte als eigenes pub/sub-Overlay gesehen werden.

Gleichverteilt über den Schlüsselraum treten Teilnehmer des pub/sub dem Pastry-Overlay bei. Von dieser Position aus können sie publishen, Topics beitreten und Topics erstellen. Um ein Topic zu erstellen, wird dem Topic eine TopicId $\in$ Schlüsselraum zugewiesen. Ein Knoten mit dem Schlüssel, der dieser Id am nächsten ist, muss ab sofort als Ausgangspunkt, also als Wurzel, des Multicastbaumes fungieren. Zu diesem Knoten wird eine entsprechende Nachricht gerouted, so dass er von seiner Aufgabe erfährt. Schon hier sieht

man in Bezug auf faire Arbeitsteilung, dass ein Knoten für ein Topic Arbeit verrichten muss, an dem er nicht unbedingt interessiert ist. Dies ist unfair, da der entsprechende Knoten Arbeit leistet, ohne davon zu profitieren

Der Aufbau des Multicastbaums ist dem Reverse Path Forwarding (RPF) aus dem ip-Multicast-Bereich abgeschaut. Hat ein Knoten Interesse an einem Topic, so sendet er eine Subscribe-Message in Richtung der Wurzel. Jeder Knoten führt eine Tabelle mit Kindern des Multicastbaumes für jedes Topic, das er kennt. Bekommt ein Knoten nun eine Subscribe-Nachricht, so schaut er, ob er schon eine Tabelle für ein entsprechendes Topic hat. Wenn ja setzt er einfach den Sender der Nachricht als Kind in diese Tabelle ein. Kennt er das Topic noch nicht, erstellt er eine Tabelle mit Kindern für das Topic, trägt dort den Sender ein und sendet für sich selbst eine Subscribe-Nachricht. Ab diesem Zeitpunkt muss er alles Interessante zu diesem Topic seinen Kindern weiterleiten. Auch hier ist jemand durch Arbeit belastet, der nicht unbedingt am Topic interessiert ist.

Unsubscriptions werden auch in Richtung der Wurzel geleitet. Möchte ein Knoten keine Ereignisse mehr zu einem Topic erhalten, so überprüft er, ob in seiner eigenen Tabelle aus Kindern niemand steht, der an diesem Topic interessiert ist. Ist dies der Fall ist, leitet er den Wunsch an seinen Vater weiter. Die Nachricht läuft solange in Richtung Wurzel bis entweder ein Knoten gefunden wird, welcher noch Interesse an einem Topic hat, bzw. noch Kinder in seiner Tabelle eingetragen sind oder bis die Wurzel erreicht ist.

In Bezug auf Churn lässt sich vermuten, dass viele Knoten, die in keiner Weise Interesse an dem Topic haben, von der unfairen Arbeitsteilung betroffen sind, da sie auf dem Weg der Subscription- bzw. Unsubscription-Nachricht in Richtung Wurzel Arbeit leisten müssen. Bei einem Pastry Overlay ohne Churn, könnte ein Knoten sehr unentschlossen sein und ständig das selbe Topic eröffnen und diesem beitreten, um kurz darauf das Topic wieder zu schließen und dieses dabei verlassen. Die Knoten, die auf dem Weg von diesem Knoten zur potentiellen Wurzel liegen, werden durch die Unentschlossenheit des einen Knoten unfairer Weise mit Mehrarbeit belastet. [22, 6]

3.3 TERA

Der Name TERA - Topic based Event Routing for peer-to-peer Architectures - sagt schon aus, dass es sich um ein pub/sub-System auf P2P-Basis handelt. Die wichtigsten Punkte, die TERA herausstellt und die das System beinhalten ist das Zusammenfassen von Knoten in ähnliche Nachbarschaften (clustering), dann dort nahezu ohne Knoten, die nicht an Themen interessiert sind, Nachrichten zu verbreiten (inner-cluster routing) und effizient von

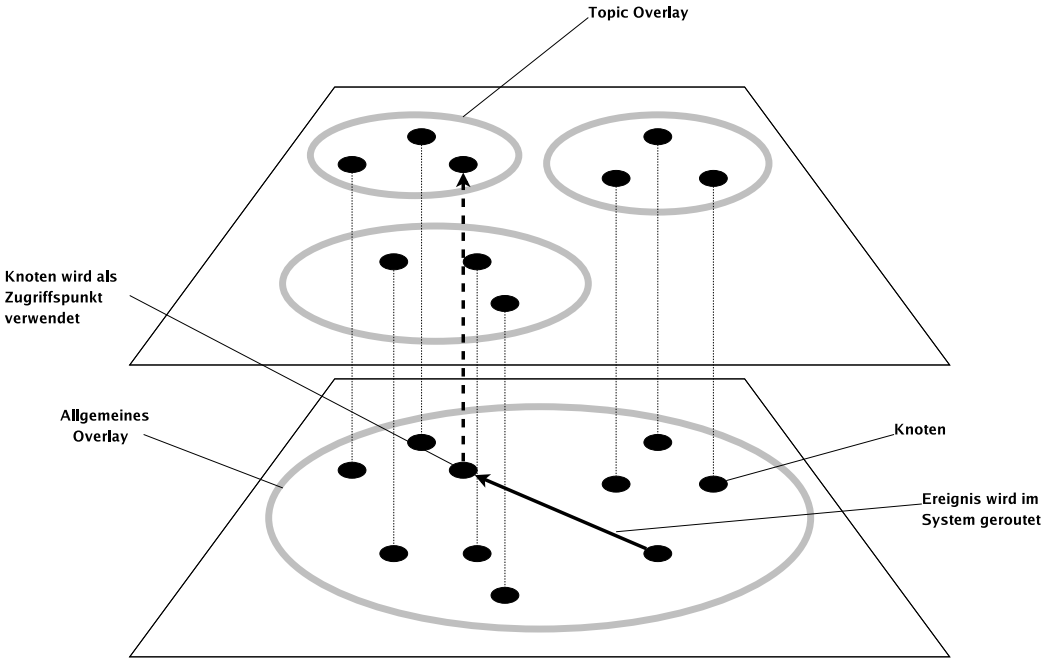

Abbildung 3.2: Architektur von TERA

außen einen Zugriffspunkt zu finden, um von außerhalb in das Topic zu publishen (outer-cluster routing).

Das System ist auf zwei Schichten verteilt (Abbildung 3.3). Die grundlegende Struktur bildet ein P2P-Overlay, in dem Gossip-Nachrichten fließen. Eine als Topic Overlay benannte Schicht bildet Gruppen aus Topics, um dort per Fluten, oder einem anderen Mechanismus, zu publishen. Der Hauptunterschied zu dem in dieser Arbeit vorgestellten System liegt darin, dass bei TERA alle Overlays unstrukturiert sind.

Ein jeder physikalische Knoten hält für jedes Topic, dem er beigetreten ist, einen virtuellen Knoten im unstrukturierten Overlay. Das Overlay Verwaltungsprotokoll sorgt dafür, dass regelmäßig Sichten, also Informationen über andere Knoten, zwischen den Peers ausgetauscht werden und so ein zusammenhängender zufälliger Graph entsteht. Da für jedes Topic ein eigenes Overlay gehalten wird, bekommt ein jedes Overlay eine Identifikation.

Um das Clustering nach einer Subscription aufrecht zu erhalten, muss ein Weg in das Cluster zu einem Zugriffspunkt des Topics für den Subscriber gefunden werden. Wenn ein Teilnehmer des Systems nicht einem Topic beigetreten ist, aber dort publishen möchte, so muss er ebenfalls einen Zugriffspunkt für dieses Topic finden. Die Aufgabe, eine Subscription oder ein

Publish außerhalb des Gebietes eines Topics an die richtige Stelle zu leiten, wird mit Hilfe zweier Methoden gelöst. Zum einen durch die in ihrer Größe beschränkte Acess-Point-Lookup-Table (APT) und zum anderen durch einen Random-Walk Algorithmus im P2P-Overlay.

Die APT könnte man als Herzstück von TERA bezeichnen. Mit dieser Tabelle ist die Möglichkeit gegeben das outer-cluster routing effizient zu gestalten und schnell einen Zugriffspunkt innerhalb des Topics zu finden. In ihr werden 2-Tupel aus KnotenID und TopicID verwaltet. Möchte man durch Einfügen eines Knotens einem bestimmten Topic beitreten wird die APT zu Rate gezogen. Möglicherweise kommt man mit nur einem hop ins Topic-Overlay. Dazu muss die Tabelle einen Eintrag für das gesuchte Topic enthalten. Gleiches gilt fürs Publishen. Hat die APT keinen entsprechenden Eintrag, so wird ein in seiner Lebenszeit beschränkter, Random-Walk durchgeführt, bis ein entsprechender Zugriffspunkt gefunden ist. Regelmäßig wird sie von Advertisments, Gossip-Nachrichten, mit neuen Informationen beliefert. Unter gewissen probabilistischen Annahmen, in denen die Topic-Beliebtheit eine Rolle spielt, werden, nachdem die APT eine gewisse Größe erreicht hat, Einträge durch die Informationen der enthaltenen Advertisments erneuert und ausgetauscht. Auch die Häufigkeit, mit der Advertisments gesendet werden, ist an der Topic-Beliebtheit festzumachen. Antiproportional zur Beliebtheit werden für ein Topic Advertisments von einem Subscriber versendet und mit genau dieser Wahrscheinlichkeit werden Einträge in der APT ausgetauscht. Die Beliebtheit wird anhand der Größe eines Topics, also der Anzahl der Subscriber, ermittelt.

Da durch parallele Subscriptions in ein Topic, welches noch nicht existiert, zwei Topic Overlays entstehen können, wurde ein Partitionsvereinigungs-Algorithmus verwendet. Anhand von unterschiedlichen Topic-Overlay Identifikationen, wird bei Advertisments festgestellt, ob zwei oder mehr Topic-Overlays des gleichen Topics bestehen. Durch Austausch von Sichten wird das System zur Partitionierung gebracht.

Im Gegensatz zu dem hier vorgestellten Ansatz wurde die Last in TERA über die Teilnehmer gleichverteilt. Es wurde in [7] hervorgehoben, dass sie dies, um hot spots zu vermeiden, als gut empfinden.

In Bezug auf faire Behandlung bei Churn lassen sich nur Vermutungen anstellen, da bisher keine Aussagen getroffen wurden. Bei hohem Churn durch Subscriptions wird die APT von anderen Teilnehmern dadurch gut besetzt sein, dass viele Advertisments aus dem gleichen Gebiet kommen und die Beliebtheit des Topics dann hoch ist. Früher oder später wird das Auffinden eines Access Points in O(1) durchgeführt werden können, da in jeder APT ein Eintrag für das Topic vorzufinden ist. Bei Churn durch Subscriptions wie durch Unsubscriptions werden schätzungsweise sehr viele veraltete Einträge

in APTs bzw. keine Einträge vorhanden sein und viele andere Knoten in anderen Topics werden durch den Random Walk betroffen werden. [7]

3.4 SpiderCast

SpiderCast ist ein Overlay, welches sich schon in der Architektur von vielen P2P-basierten pub/sub-Systemen abhebt. Hier wird kein eigenes Topic-Overlay aufgebaut, sondern es werden innerhalb des unstrukturierten P2P-Systems Knoten mit gleichem Interesse verbunden. Es entstehen viele zusammenhängende Teilgraphen zwischen den Teilnehmern des Systems, wobei jedes Topic einen Teilgraphen stellt. Durch diese Teilgraphen wird ein Topic-clustering erreicht.

Ein Teilnehmer des pub/sub-Systems ist mit einem physikalischen Knoten im System vertreten. SpiderCast legt besonders Wert darauf, einen kleinen Ausgangsgrad des einzelnen Knoten im Overlay zu halten und trotzdem einen geringen Durchmesser der Teilgraphen des P2P-Systems zu bekommen. Geringer Ausgangsgrad bedeutet, dass so wenige Nachbarn wie möglich in Nachbarschaftstabellen gehalten werden. Diesen erreichen sie durch Ausnutzung von lokalen Gegebenheiten. Man beruft sich darauf, dass zwei Knoten der selben Nachbarschaft in zwei oder mehr gleichen Topics vertreten sind. Demzufolge reicht es nur eine Verbindung und nur einen timeout für die Verbindung zu halten. Zusätzlich sollen möglichst $K \in \mathbb{N}$ Nachbarn, die ebenfalls am selben Topic Interesse zeigen, für jedes Topic gehalten werden. Eine maximale Verbindungszahl ist durch $K* \mid Topics \mid +\varepsilon$, wobei ε eine kleine Konstante darstellt, gegeben. So erreicht SpiderCast, dass der Ausgangsgrad der Knoten nicht linear, so wie z.B. bei Scribe [6], sondern logarithmisch wächst.

Bei SpiderCast wird der Versuch unternommen, sich so effektiv wie möglich von Nachbarn zu trennen sobald K oder eine maximale Verbindungszahl überschritten ist. Es wird an die Nachbarn eine Disconnect-Nachricht gesendet, für die der Effekt des Trennens am wenigsten Auswirkung auf ihre K-Nachbarschaft hat.

Wichtig ist die *interest-view*, die durch ein zufallsbasiertes Mitgliedschaftsprotokoll gehalten wird. Darin werden Knoten mit Informationen zu deren Identifikation und zu deren Subscriptions zu Topics gehalten. Subscribt nun ein Knoten zu einem bestimmten Topic, so kann er, falls er noch nicht mit genug anderen Knoten, welche das selbe Interesse an einem Topic haben, verbunden ist, sich aus der *interest-view* bedienen und Verbindungsgesuche an einen passenden Knoten mit entsprechendem Interesse aus dieser Liste senden.

Welcher Knoten als Nachbar in die Nachbarschaftstabelle eingetragen wird, ist mit Hilfe von zwei alternativen Heuristiken bestimmbar. Eine Random-Heuristik wählt zufällig Knoten, die ihr Interesse bei mindestens einem Topic teilen, als Nachbar. Eine Greedy-Heuristik wählt einen Knoten als Nachbarn, der möglichst viele Topics mit dem Knoten, welcher sich entscheidet, gemeinsam hat. In der Nachbarschaftstabelle werden Informationen über die Identifikation und die Topics, welchen ein Nachbar beigetreten ist, gehalten.

Empfängt ein Knoten ein Verbindungsgesuch, so entscheidet er anhand der maximalen Verbindungszahl, ob er diesen Knoten in die Nachbarschaftstabelle aufnimmt oder nicht. Würde diese mit dem neuen Nachbarn überschritten, so sendet ein Knoten eine Redirect-Nachricht zurück. Darin wird der anfragende Knoten aufgefordert sich zu einem anderen, in der Nachricht mitgelieferten, Knoten zu verbinden. Der mitgelieferte Knoten ist ein Knoten aus der Nachbarschaftsliste des Senders der Redirect-Nachricht, welcher noch Ressourcen frei hat. Informationen über Knoten, von denen ein Knoten weitergeleitet wurde, hält dieser in einer speziellen Liste und kann sie für die Verbindungs-Gesuche mit einbeziehen. [5]

Kapitel 4

Churn

4.1 Churn in Publish/Subscribe Systemen

Bei *Churn* handelt es sich um ein aus Change und Turn zusammengesetztes Kunstwort. In der Wirtschaftswissenschaft bezeichnet es die Kundenzu- und abwanderung von Unternehmen. [23]

Im Sinne der Verteilten Systeme bezeichnet es die Häufigkeit, mit der ein Prozess einem System beitritt bzw. es verlässt. Speziell für Publish/Subscribe Systeme bedeutet es, wie häufig Subscriptions bzw. Unsubscriptions durchgeführt werden. Wie bei P2P-Systemen kann bei Veteilten Systemen im Allgemeinen davon ausgegangen werden, dass alle Entitiäten das gleiche Interesse an einem solchen System haben. Deshalb wird oft die Last, das System aufrecht zu erhalten, und die Kosten für Churn gleich über das System verteilt.

In Publish/Subscribe Systemen dagegen ist die Wahrscheinlichkeit für ein gleichverteiltes Interesse an allen Ereignissen gering. Stattdessen wird es den Fall geben, dass ein Topic im Vergleich zu anderen interessanter erscheint und deshalb der Churn durch ständiges Beitreten und Verlassen darin groß ist. Währendessen können andere Topics nahezu keine Subscriptions erhalten, da sie uninteressant sind. Deshalb werden in dieser Arbeit Topic-basierte Publish/Subscribe Systeme auf Churn untersucht. Es wurden Methoden entwickelt die gewährleisten, dass diejenigen Prozesse, welche an Themen mit geringerem Churn Interesse haben, nicht allzuviel Mehrarbeit leisten müssen als Teilnehmer eines Topics mit mehr Churn.

Es ist ein kleiner, aber nicht zu unterschätzender Unterschied, wodurch Churn verursacht wird. Churn kann nur durch Subscriptions, nur durch Unsubscriptions oder durch beide Operationen gemeinsam verursacht werden. Diese Unterscheidung ist wichtig, da für beide Aktionen unterschiedliche

Schritte durchgeführt werden müssen. Wenn man die Mehrarbeit anderer Topics untersuchen möchte, so muss man genau wissen, wo diese zusätzlich durch Subscriptions und Unsubscriptions arbeiten müssen.

Wie im vorherigen Kapitel über verwandte Arbeiten zu sehen ist, werden Knoten, die das gleiche Interesse haben, also dem gleichen Topic beigetreten sind, oft in gleichen Gebieten zusammgefasst. Diese Gruppierung geschieht, damit kaum Nachrichten-Overhead beim Publishen entsteht. Ebenso existieren Knoten, meist von Subscribern gestellt, die als Einstiegspunkte in diese Gebiete dienen. Bei GosSkip wären dies alle Knoten, die das Topic aufspannen. Hingegen bei Scribe sind dies alle Knoten, welche am Multicastbaum beteiligt sind. Meist muss eine lookup-ähnliche oder eine join-Nachricht, wie in Scribe oder GosSkip, gesendet oder ein Random-Walk, wie im worst-case von TERA, durchgeführt werden, um Informationen über einen Einstiegspunkt zu erhalten. Da dabei die Knoten in Gebieten von Topics, für die die Subscriptions nicht von Interesse sind, betroffen sind, ist dies Arbeit, die andere Knoten für die Subscriptions in ein Gebiet mit Churn mittragen.

Unsubscriptions werden häufig dadurch realisiert, dass ein Knoten einfach das System verlässt, so wie in GosSkip oder in TERA. Dies wäre entsprechend einem Ausfall zu behandeln. Eine andere Möglichkeit ist es spezielle Unsubscribe-Nachrichten durch das System zu routen, wie bei Scribe. Allgemein kann man beobachten, dass diese Änderungen meist lokal sind und seltener Knoten in anderen Topics belasten. Der Vorteil, von welchem man hierbei profitiert, ist, dass Elemente mit gleichem Interesse im selben Cluster zu finden sind.

Churn, der durch beides, Subscriptions wie Unsubscriptions, verursacht wurde, summiert Kosten, die durch diese Operationen verursacht wurden, auf. Wie dargelegt, sind vorallem die Kosten für Subscriptions in anderen Topics teuer und Unsubscriptions verursachen nur lokal Kosten. Beides zusammen kann in TERA dazu führen, dass APTs keine richtigen Informationen halten. GosSkip kann, wie im Evaluierungskapitel gezeigt wird, zusammenbrechen, da nie lange Verbindungen aufgebaut werden, wenn der Churn zu hoch ist. Diese langen Verbindungen werden allerdings für die Subscription benötigt.

4.2 Fairness-Metrik bezüglich Churn

Fairness definiert sich als die Verteilung der Arbeit anhand des Nutzens eines einzelnen Teilnehmers am System. Es kann deshalb Teilnehmer geben, welche mehr Arbeit verrichten als andere. Damit steht Fairness im Kontrast zur Lastverteilung, bei der die Arbeit gleichmäßig über alle Teilnehmer ver-

teilt wird. Damit Fairness erreicht wird, muss die Arbeit proportional zum Nutzen verteilt werden. In dieser Arbeit werden Subscriptions sowie Unsubscriptions als Nutzen angesehen. Deshalb wird Churn als Maß für den Nutzen verwendet.

Churn kann durch Subscriptions zu bestimmten Topics, Arbeit in anderen Topics, mit weniger Churn, verursachen. Es wurde erläutert, dass diese Arbeit vorallem durch lookup-ähnliche Nachrichten, join-Nachrichten oder sonstige Suchen nach Informationen über Einstiegspunkte andere Topics belastet. Ein Verfahren ist fair, wenn die Arbeit, Suchanfragen weiterleiten zu müssen, sich proportional zum Churn pro Topic verhält. Lookup ist von nun an im Sinne einer allgemeinen Suche nach den Einstiegspunkten im System festgelegt.

Möchte man nun die Arbeit so fair verteilen, dass jedes Topic soviel Last tragen muss, wie es der Churn vorgibt, bietet die folgende Metrik die Möglichkeit auszuwerten, wie fair ein Verfahren ist. Im Folgenden sei eine endliche Teilmenge an Identitäten $I \subseteq \mathbb{N}$ für Schlüsselwörter T von Topics gegeben. Damit eine eindeutige Ordnungsrelation für Topics existiert, wird das Schlüsselwort durch eine Hashfunktion h auf die Identitätenmenge abgebildet h(T) = I. Hierbei ist i die größte Identität der Menge. Für einzelne Identitäten k $\in I$, sei $churn_k$ der Churn im Topic mit Identität k. Die Anzahl der lookup-Weiterleitungen bzw. join-Weiterleitungen können dementsprechend für jedes Topic mit Identität k als $lookup_k$ bzw. $join_k$ definiert werden.

$$\frac{churn_0 + 1}{lookup_0 + join_0 + 1} \approx \frac{churn_1 + 1}{lookup_1 + join_1 + 1} \approx ... \approx \frac{churn_i + 1}{lookup_i + join_i + 1}$$

Die einzelnen Quotienten beschreiben das Verhältnis von Churn in einzelnen Topics, zur Arbeit dieser Topics, Suchanfragen nach Einstiegspunkten weiterzuleiten. Da sich die Arbeit proportional zum Churn verteilen soll, müssten sich die einzelnen Quotienten angleichen. Dadurch kann ermittelt werden, wie fair ein Verfahren ist, indem bestimmt wird, wie stark die Quotienten voneinander abweichen.

Die einzelnen Quotienten werden fortan als Fairnessquotienten bezeichnet. Von einem fairen Verfahren spricht man, wenn die maximale Differenz der einzelnen Fairnessqotienten gegen null strebt, denn dann leistet jedes Topic seinem Churn entsprechend Arbeit. Ein Verfahren ist fair und produziert kaum zusätzliche Arbeit durch lookup-Nachrichten, wenn der Fairnessquotient gegen eins strebt. Dies würde bedeuten, dass für jede Subscription genau ein hop für join oder lookup benötigt würde. Schlecht ist in diesem Sinne, wenn der Fairnessquotient gegen null strebt, denn dann ist ein Gebiet eines Topics durch sehr viel Arbeit für andere Topics belastet oder die Suche nach Einstiegspunkten benötigt zu viele hops.

Mit der vorgestellten Metrik ist es möglich, sowohl Aussagen über Fairness, als auch Aussagen darüber, mit welchen Kosten diese Fairness erreicht wird, zu treffen. Fairness sollte nicht um jeden Preis erreicht werden. Wenn durch den Versuch, Fairness herzustellen, zusätzliche Arbeit entsteht und dadurch der definiert Fairnessquotient immer weiter gegen null strebt, so erreicht man nicht die geforderte Entlastung von Topics mit geringerem Churn. Ein Verfahren, welches Fairness herstellt, sollte erreichen, dass unfair behandelten Topics Arbeit abgenommen wird, damit deren Fairnessquotient steigt. Topics hingegen, bei denen hoher Churn entstanden ist, müssen mehr Arbeit erhalten. Dadurch sinkt deren Fairnessquotient.

4.3 Konzeption des Overlays

Anforderungen. Um Fairness gewährleisten zu können, muss ein P2P-Overlay, als Grundlage für ein dynamisches pub/sub - System, gewisse Eigenschaften erfüllen. Diese Eigenschaften sind erfüllt, wenn mit Hilfe des Overlays gezielt Arbeit verteilt werden kann. Wie in vorherigen Kapiteln erwähnt, werden zusammengehörige Topics häufig in gleichen Gebieten zusammengefasst, um keinen Overhead durch das Publishen zu erzeugen. Suchen nach Einstiegspunkten in diese Gebiete verursacht Arbeit in jedem Gebiet eines Topics, das auf dem Weg in das Zielgebiet getroffen wird.

Ein jeder Knoten im P2P-System kann als Einstiegspunkt und als Arbeitseinheit gesehen werden. Jede Arbeitseinheit muss eine Grundlast an regelmäßiger Arbeit verrichten, um das System aufrecht zu erhalten, lange Verbindungen zu verwalten und um Nachbarschaftslisten zu aktualisieren. Die verteilte Suche zu unterstützen, indem bei Bedarf Nachrichten weitergeleitet oder beantwortet werden, ist ebenfalls Teil ihrer Aufgabe. Von einem fairen Verhalten spricht man, wenn auch die Arbeit, Suchanfragen weiterleiten zu müssen, gleich über die Knoten verteilt ist. Dabei würde jede Arbeitseinheit gleichviel Kosten tragen.

Die Wahl der Identitäten, jedes einzelnen Knotens im System, ist Entscheidend, wenn man die Einstiegspunkte innerhalb des Topics gleichverteilen möchte. Damit jeder Knoten innerhalb des Gebietes eines Topics gleichverteilt gesucht werden kann, müssen die einzelnen Identitäten darin gleichverteilt werden. Diese Identitäten müssen dann bei der Suche ebenfalls gleichverteilt gewählt werden. Diese Anforderungen sind durch das Prinzip einer DHT erfüllt.

Ein wünschenswertes Verhalten wäre gegeben, wenn ein Gebiet, gleichverteilt auf jeden Knoten darin, soviel Arbeit leisten müsste, wie Suchanfragen in dieses Gebiet gehen. GosSkip bietet die Möglichkeit Arbeit durch

Veränderung der Größe von Gebieten zu verteilen. Diese Eigenschaft von GosSkip ist der gleichmäßigen Verteilung aller Verbindungen über alle Knoten, in einem, zu einem Konstrukt, ähnlich einer Perfekten Skipliste, konvergierten Zustand, zuzuschreiben.

GosSkip im Vergleich. Sei N die Anzahl der Knoten im GosSkip-Ring, so besitzt jeder Knoten $2 * log(N)$ Verbindungen und ebensoviele zeigen auf den Knoten selbst. Schaut man sich dagegen Chord [15] an, so sind dort lange Verbindungen in einer Fingertable genannten Nachbarschaftsliste gespeichert. Es werden dort Nachbarn eingetragen, deren Schlüssel 2^0, 2^1 etc. vom Schlüssel des Besitzers der Fingertable entfernt liegen. Wenn ein Knoten einen großen Bereich an Schlüsseln zu verwalten hat, so kann es passieren, dass sehr viele lange Verbindungen auf diesen zeigen. Bei einem kleinen Bereich ist es gerade umgekehrt. Damit wären die Verbindungen nicht gleichmäßig über alle Knoten verteilt. Bei DHT -Strukturen wie Chord, sind nur bei einer sehr guten Verteilung der Schlüssel auch die langen Verbindungen gut verteilt. Ist die Verteilung der Schlüssel schlecht gewählt, wie beispielsweise geschehen kann, wenn man spezielle Schlüssel wählt und nicht, wie es meist der Fall ist, eine zufällige Verteilung der Schlüssel vorgenommen wird, dann kann es passieren, dass auf einen einzigen Knoten alle langen Verbindungen zeigen. Das gerade beschriebene Prinzip ist beispielhaft in Abbildung 4.1 dargestellt.

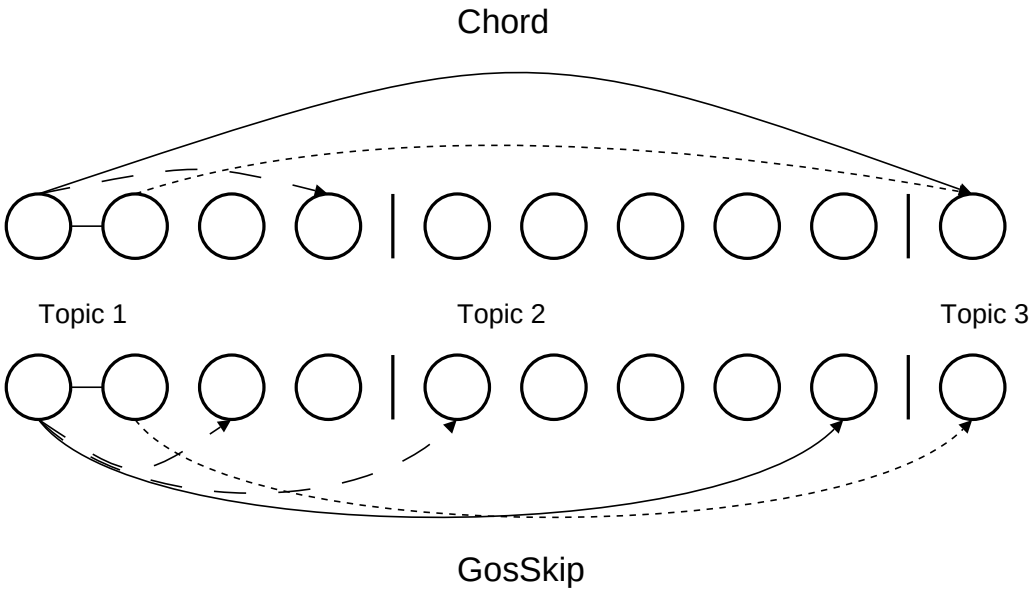

Abbildung 4.1: Exemplarischer Vergleich von Chord und GosSkip Nachbarschaftstabellen

Durch diese Eigenschaft des Gleichverteilens brilliert GosSkip. Bei verschieden großen Topic-Bereichen ist die Anzahl der langen Verbindungen, die in das Gebiet reichen, größer. Dies wirkt sich insbesondere auf die Suche und den Beitritt ins System aus, denn je größer ein Gebiet eines Topics, umso häufiger wird es bei gleichmäßig verteilten Startpunkten für Suchanfragen getroffen. Und deshalb ist hier die Motivation gegeben, ein System aufzubauen, das adaptiv an Churn angepasst die Bereiche vergrößert und damit ebenso die Wahrscheinlichkeit, dass diese Bereiche getroffen werden. Entsprechendes soll durch das verkleinern von Bereichen erreicht werden.

Modellieren der Größe von Gebieten der Topics. Ein auf GosSkip basierendes pub/sub-System muss also die Möglichkeit schaffen, durch gezieltes Hinzufügen und Entfernen von Knoten in speziellen Bereichen deren Größe zu variieren. Dies brachte die Idee hervor, spezielle Knoten zu erzeugen, die ausschließlich die Aufgaben übernehmen, Gebiete zu vergrößern und dort für eine bestimmte Zeitspanne zu bleiben. Diese zusätzlichen Knoten haben den Namen Churnnode bekommen. Sie übernehmen nur im P2P-System Arbeit und liefern deshalb einen Einstigespunkt aus ihrer direkten Nachbarschaft zurück. Jeder andere Knoten im System, welcher zusätzlich für das pub/sub-System Arbeit leistet, wird Workingnode genannt. Diese Workingnodes sind demzufolge die erwähnten Einstiegspunkte. Da GosSkip virtuelle Knoten und nicht physikalische Knoten - Recheneinheiten - vorsieht, können mehrere Working- und Churnnodes auf einem physikalischen Knoten im System vorhanden sein. Die Last, welche durch den join-Prozess des Churnnode entsteht, wird durch dessen Nutzen, das System fairer zu gestalten, aufgewogen.

4.4 Dynamische Anpassung

Die Problematik, die hier behoben werden soll, ist es, unnötige Arbeit von Knoten in anderen Gebieten, als dem mit hohem Churn, zu nehmen. Um diese Problematik zu lösen, muss ein Knoten mit Hilfe des lokal gemessenen Churns Aussagen treffen können, anhand denen er dieser unfairen Behandlung entgegen wirken kann. In dezentralen Architekturen hat jedoch weder ein virtueller noch ein physikalischer Knoten die globale Sicht. Deshalb werden nun zwei Verfahren vorgestellt, dennoch durch Churn verursachte Arbeit zu verteilen. Eines der Verfahren versucht rein lokal die Situation zu verbessern, das andere versucht den Knoten eine angenäherte globale Sicht zu verschaffen und an Hand dieses Wissens zu adaptieren.

Verfahren 1: First come, first serve

Um vernünftige Aussagen über Churn treffen zu können, muss dieser zuerst gemessen werden. Dazu kann jeder Zugriffspunkt in das Gebiet eines Topics zählen, wie häufig er für eine Subscription verwendet wurde, bzw. wie häufig er Unsubscriptions mitbekommen hat. Dies wird über ein zeitliches Intervall fester Größe, oder auch Runde genannt, gemessen. Übersteigt der Wert im nächsten Intervall den Wert des vorherigen, so kann man davon sprechen, dass die Rate des Churns gestiegen ist. Dies kann als Indikator herangezogen werden, ab wann entsprechende Maßnahmen gegen den hohen Churn eingeleitet werden müssen. Diese Aussagen kann ein Knoten lokal treffen, wenn die Subscriptions sowie Unsubscriptions gleichverteilt über das gesamte Gebiet des Topics verteilt sind. In diesem Fall wären die ermittelten Werte repräsentativ.

Churnnodes sind als Arbeitseinheiten aufzufassen, welche gezielt eingesetzt werden, um Gebiete zu vergrößern. Der erste Vorschlag, dem Churn durch Churnnodes gerecht zu werden, befasste sich damit, Knoten, die gerade einem Topic beitreten wollten, zu bestrafen, indem sie einen oder mehrer Churnnodes stellen mussten. Dies wurde als fair erachtet, da derjenige, der Churn verursacht hat damit bestraft würde eine Arbeitseinheit, also einen Churnnode, ins System zu bringen. Beim Beitritt in das System bekommt ein Knoten, wenn ein Einstiegspunkt, also ein Workingnode, entschieden hatte, es wäre mehr oder gleichviel Churn als in der Runde zuvor, eine Nachricht. In dieser wird der Knoten aufgefordert dem System mit einer bestimmten Anzahl Churnnodes, die die Arbeitseinheiten darstellen, beizutreten und diese für eine gewisse Rundenanzahl im System zu halten.

Nicht jeder Subscriber kann bei zu hohem Churn durch Subscriptions einen oder mehrere Churnnodes einfügt, denn das System würde dann kollabieren. Das Verhalten liegt darin begründet, da bei hohem Churn sehr viele Subscriptions kurz hintereinander eingehen. Dementsprechend werden Churnnodes kurz hintereinander eingefügt. Diese werden dann auch kurz hintereinander aus dem System genommen. Wenn also die Churnnodes einen zu großen Anteil $y \in \mathbb{N}$ am System haben und zu schnell gehen, dann kann keine k-Ausfallsicherheit der Struktur mehr greifen, sei $k \in \mathbb{N}$ und $y > k$. Es sollte also dafür gesorgt werden, dass nicht zu viele Knoten auf einmal das System verlassen.

Zwei verschiedene Ansätze dies zu beheben sind durch postventive und präventive Methoden gegeben. Postventiv wäre ein Ansatz, der, wenn der Knoten geht, anhand der Systemgröße eine Wahrscheinlichkeit bestimmt, mit der ein Churnnode gehen darf. Präventiv wäre ein Ansatz, der im Vorhinein schon bestimmt, dass schon mehr als genug Knoten eingefügt wurden und

deshalb keine, oder anhand einer Wahrscheinlichkeit, Churnnodes einfügen lässt. Präventive Ansätze haben den Vorteil, dass nicht zu viel zusätzliche und unnötige Last durch den join-Prozess eines Churnnodes verursacht wird.

Es konnte gezeigt werden, dass sich eine Verbesserung durch die Verdoppelung des Gebietes erreichen lässt. So reicht es bei einem präventiven Ansatz, dass jeder Einstiegspunkt in das Gebiet eines Topics dafür sorgt, dass aus der Menge, der von ihm eingefügten Knoten nur einer aktiv einen Churnnode zu einem Zeitpunkt stellt.

Verfahren 2: Ein Fenster zur globalen Sicht

Verbesserungen zu Verfahren 1. Die gerade erwähnte Methode zum Umgang mit Churn birgt einige Probleme. Einerseits hat hier jeder Einstiegspunkt nur eine lokale Sicht. Ein Gebiet, dessen Rate an Churn zehn mal kleiner ist als die Rate eines anderen Gebietes, fügt dennoch die gleiche Anzahl an Churnnodes ein. Dem Prinzip, dass sich die Größe des Gebiets eines Topics an die Suchanfragen in dieses Gebiet anpassen muss, damit sich die Suchanfragen gleichmäßig über die Knoten verteilen, passt es sich also nicht an.

Das zweite Problem, dass das oben erwähnte Verfahren besitzt, liegt in der Instabilität der Churnnodes. Zum einen sind diese nur für ein bestimmtes Intervall im System und zum anderen werden sie von Knoten gestellt, die Churn verursacht haben. Demzufolge ist auch nicht unbedingt sichergestellt, dass diese Knoten im System bleiben. Bei dieser Frage nach Stabilität geht man davon aus, dass ein Knoten, der schon länger vom System profiertiert, dies noch länger tun möchte und daher stabiler ist. Bezieht man sich dabei auf einen Fairnessaspekt, so kann man argumentieren, dass der Knoten, welcher am längsten vom System profitiert mit einer Arbeitseinheit bezahlen muss, um dies auch weiterhin zu dürfen. Deshalb ist es sinnvoll den stabilsten Knoten, welcher von einem Einstiegspunkt in das Topic eingefügt wurde, als Churnnode zu verwenden.

Mit Churnnodes kann man nur durch Hinzufügen wirkungsvoll Effekte erzielen, denn die Anzahl der Workingnodes geben eine untere Schranke für die Größe eines Gebietes an. Durch Zufall kann in einem Gebiet, welches um den Faktor hundert kleiner ist als ein anderes, der Churn hundert mal so groß wie in dem anderen Gebiet sein. Wie zuvor schon diskutiert ist es nicht sinnvoll so viele Churnnodes in das System zu bringen, dass dieser Churn ausgeglichen wird, da alle diese Knoten auch wieder gehen. Der Gedanke, um dies zu lösen war, normale Zugriffspunkte, also Workingnodes, anstatt Churnnodes einzufügen und diese so zu wählen, dass man von einer gewissen Stabilität bei ihnen ausgehen kann.

Anfragen. Bei dem im Folgenden vorgestellten Ansatz geht es darum, den prozentualen Anteil eines Topics am Gesamtchurn auf den Anteil der Workingnodes eines Topics an der Gesamtzahl der Knoten eines Systems abzubilden. Dabei werden Arbeitseinheiten, also Workingnodes, fair anhand des Churns verteilt. Der verteilte Algorithmus 1 wird auf jedem Workingnode ausgeführt.

Um die Adaption durchführen zu können beschafft sich das nächste Verfahren eine der globalen Sicht angenäherte Sicht. Dies wird erreicht, indem mit gezielten Anfragen, Stichproben von anderen Arbeitseinheiten abgefragt werden. Hierzu kann man Vorteile des GosSkip-Overlay ausnützen. Da eine lange Verbindung auf Ebene i $\in \mathbb{N}$ genau zum 2^i-ten Nachbarn zeigt, kann man die Größe n $\in \mathbb{N}$ des Gesamtsystems auf $2^{max} < n < 2^{max+1}$ abschätzen, sei max die Höhe der Nachbarschaftsliste. Es wird vorausgesetzt, dass keine Verbindungen, die mehr Knoten überspringen als es Teilnehmer am System gibt, existieren. Dementsprechend kann man sich mit einer Suche nach seinem x-ten Nachfolger, sei x $\in \{l \mid l \in \mathbb{N} \wedge 1 \leq l \leq n\}$, gleichverteilt über das System Informationen über Knoten und deren Churn beschaffen. Ein jedes Topic ist durch einen Hashwert über dessen Schlüsselwort bestimmt. deshalb sei t, t1, t2, ... tm $\in$ der Menge der Hashwerte, wenn m die Kardinalität der Menge ist. Für den nun vorgestellten Ansatz muss ein Knoten andere Knoten über deren Rate an Churn $c_t \in \mathbb{N}$ und einer Abschätzung der Größe $g_t \in \mathbb{N}$ ihres Topic t, also die Anzahl der Knoten in deren Gebiet, abfragen. Bei den Anfragen wird davon ausgegangen, dass durch eine Gleichverteilung der Subscriptions über alle Knoten in dem Gebiet eines Topics, der Churn lokal bei jedem Knoten innerhalb des Gebietes gleichverteilt ist und deshalb auch repräsentativ für jedes Topic ist.

Periodisch werden nun Anfragen gestellt und Antworten gegeben. Eine zur Größe des Systems skalierbare Anzahl angefragter Knoten wären logarithmisch viele, was genau der Höhe einer Nachbarschaftsliste entspricht.

Einschätzung ermitteln. Für jeden abgefragten Wert hält ein Knoten die Information welches Topic t abgefragt wurde und ein 2-Tupel (c_t, g_t). Anhand dieses Wissens trifft ein Workingnode Aussagen über ein Topic, für welches er arbeitet. Da verschiedene Workingnodes innerhalb des gleichen Topic zufälligerweise angefragt werden können, sollten zuerst die Werte $c_{t1}, c_{t2}, ...c_{tn}$ sowie $g_{t1}, g_{t2}, ..g_{tn}$, falls t1 = t2 = ... = tn und n $\in \mathbb{N}$ die Anzahl der Tupel aus dem gleichen Topic, gemittelt werden. $c_t = \frac{c_{t1}+c_{t2}+..+c_{tn}}{n}$ bzw. $g_t = \frac{g_{t1}+g_{t2}+..+g_{tn}}{n}$. Diese gemittelten Werte ersetzen nun die einzeln abgefragten Werte.

Als nächstes werden der Gesamtchurn c_{gesamt}, sowie die Gesamtzahl der Knoten g_{gesamt} aus dem gemittelten Werten berechnet. Der Gesamtchurn

c_{gesamt} ist die Summe aller c_t. Die Anzahl der Knoten g_{gesamt} ist dementsprechend die Summe aller g_t. Mit dem Gesamtchurn kann ein Knoten prozentual einschätzen, wie sich der lokale Churn im Vergleich zum globalen Churn verhält. Dieser prozentuale Anteil p_c ergibt sich durch $\frac{c_{lokal}}{c_{gesamt}+1}$. Da mit hoher Wahrscheinlichkeit nicht alle Topics durch die Anfragen getroffen werden, ermittelt man eine angenäherte globale Sicht auf g_{gesamt} Knoten. Die Anzahl der Knoten $num \in \mathbb{N}$, welche in einem Topic gestellt werden müssten, um dem Anteil am Gesamtchurn gerecht zu werden, ergibt sich zu $num = p_c * g_{gesamt}$.

Da schon eine lokal angenäherte Menge an Knoten g_{lokal} in einem Topic gestellt werden, ergibt sich ein Korrekturwert k $\in \mathbb{N}$ zu $num - g_{lokal}$. Dieser Korrekturwert gibt an um welche Anzahl an Knoten insgesamt ein Topic verändert werden soll. Da diese Anzahl auf alle Workingnodes im selben Topic verteilt werden soll, werden nur anhand einer Wahrscheinlichkeit weitere Knoten eingefügt oder entfernt. Ist k positiv, so müssen eventuell Knoten hinzugefügt werden. Dementsprechend müssen bei einem negativen k-Wert eventuell Knoten aus dem Topic entfernt werden.

Probabilistisches Einfügen und Entfernen. Um die Reaktionen nicht zu extrem für das System zu gestalten und da wahrscheinlich g_{lokal} andere Workingnodes die gleiche Entscheidung getroffen haben, werden Knoten nur anhand einer Wahrscheinlichkeit eingefügt bzw. entfernt. Die Entscheidung, ob ein Knoten eingefügt wird, geschehen in Topics mit höherem Churn und deshalb treten neue Knoten genau diesem Topic als Workingnode bei. Äquivalent verlasen Knoten Gebiete mit niedrigem Churn.

Für das Einfügen hat sich die Wahrscheinlichkeit $\frac{k}{g_{lokal}}$ als praktikabel herausgestellt. Diese sorgt dafür, dass bei großem k der ermittelte Wert num sehr schnell angenähert wird. Damit bei kleinem k nicht zu viele Knoten eingefügt werden, um num nicht zu Überschreiten, ist dann die Wahrscheinlichkeit geringer mit der ein Knoten hinzugefügt wird. Würde man mit der gleichen Wahrscheinlichkeit Knoten aus dem System nehmen, wäre dies katastrophal. Wäre kein Churn vorhanden, die Rate c_{lokal} ist also null, würde der prozentuale Anteil p_c sich ebenfalls zu null ergeben. Damit wäre $k = -g_{lokal}$ und damit $\frac{|k|}{g_{lokal}} = 1$. Dies würde dazu führen, dass das Topic sofort aussterben würde. Deshalb wurde die Wahrscheinlichkeit, mit der Knoten das System verlassen, auf $\frac{|k|}{g_{gesamt}}$ eingestellt. Dabei wird davon ausgegangen, dass g_{gesamt} entsprechend größer als g_{lokal} ist. Um einem Aussterben eines Topics vorzubeugen ist eine Beschränkung sinnvoll, welche einen Workingnode nicht aus dem System nimmt, wenn er alleine für das Topic verantwortlich ist.

// tupelSet ≡ Menge aller erhaltenen 2-Tupel (c_t, g_t)
$tupelSet = \emptyset$;
c_{lokal} ; // lokal ermittelter Churn
g_{lokal} ; // lokal eingeschätzte Größe eines Topics
ownTopic ; // Schlüsselwort für das Topic, indem dieser Knoten arbeitet

UPON Receive (Samplerequest)
  send (Sampleresponse, c_{lokal}, g_{lokal}, destination: anfragender Knoten);

UPON Receive (Sampleresponse, c_i, g_i)
  tupelSet= tupelSet $\bigcup \{(c_i, g_i)\}$;

UPON TIMER
  for *0 .. Höhe der Nachbarschaftsliste* **do**
    send (Samplerequest, destination: zufälliger Zielknoten);
  end
  $c_{gesamt}, g_{gesamt} = 0$; // Gesamtchurn und Gesamtzahl der Knoten
  $c_{t1}, g_{t1}..c_{tm}, g_{tm} = 0$; // Churn und Knotenzahl pro Topic bei m abgefragten Topics mit Identitäten t1 .. tm
  k = 0 ; // Korrekturwert
  tupelSet= tupelSet $\bigcup \{(c_{lokal}, g_{lokal})\}$;
  forall *(elements* $e \in tupelSet$*)* **do**
    $c_{Topic(e)} = c_{Topic(e)} + e.c$;
    $g_{Topic(e)} = g_{Topic(e)} + e.g$;
    $tupelSet = tupelSet - \{(e.c, e.g)\}$;
  end
  for $i = t1..tm$ **do**
    // Mitteln aller abgefragten Werte c_i, g_i da mehrmals Knoten aus dem gleichen Topic abgefragt wurden
    $c_{gesamt} = c_{gesamt} + gemitteltes\,(c_i)$;
    $g_{gesamt} = g_{gesamt} + gemitteltes\,(g_i)$;
  end
  $k = \left(\frac{c_{lokal}}{c_{gesamt}+1} * g_{gesamt}\right) - g_{lokal}$;
  if $k > 0$ **then**
    Füge einen Knoten mit Wahrscheinlichkeit $\frac{k}{g_{lokal}}$ zu ownTopic mit zufälliger Position ein;
  end
  if $k < 0$ **then**
    Lösche diesen Knoten mit Wahrscheinlichkeit $\frac{|k|}{g_{gesamt}}$;
  end

Algorithmus 1 : Algorithmus für Verfahren 2

Kapitel 5

Konzepte für ein dynamisches pub/sub System

5.1 Grundsystem

Das hier implementierte und verwendete Topic-basierte pub/sub-System setzt auf eine Implementierung von GosSkip auf. Das GosSkip-Overlay wird zur Anordnung beteiligter virtueller Knoten, zum Informationsaustausch der virtuellen Knoten und zur Suche spezieller virtueller Knoten verwendet. Zur Erinnerung - Peers oder (virtuelle) Knoten sind im Sinne von GosSkip solche Knoten, die sich mannigfaltig auf verschiedenen physikalischen Knoten, also einer einzigen Recheneinheit, befinden. Das aufgesetzte pub/sub-System muss im Wesentlichen gezielt Knoten in dem GosSkip-Overlay einfügen, Subscriptions bearbeiten, Publishing ermöglichen und Benachrichtigungen erhalten und ausliefern.

Die wichtigste Eigenschaft, welche dieses System erfüllen muss, ist faire Lastverteilung [8, 24] und Fairness unter Churn zu ermöglichen. Dabei ist der Systemaufbau entscheidend. Der einfachste Ansatz hier ein Topic-basiertes pub/sub-System aufzubauen wäre es, einen Teilnehmer als virtuellen Knoten zu jedem Topic, welches ihn interessiert, in den GosSkip-Ring einzufügen. Sinnvoll ist es Subscriber mit ähnlichem oder gleichem Interesse in die selbe Nachbarschaft zu bringen, da sonst zu viele Knoten, die nicht an einem Topic interessiert sind, beim Publishen arbeiten müssten. Deshalb könnte hierbei die Sortierung der Namen längenlexikographisch anhand des Schlüsselworts geschehen. Publishing wäre mit Hilfe des in [2] vorgestellten Spreading Algorithmus möglich.

Eine solch simple Lösung wurde nicht gewählt, um faire Lastverteilung und Churn beachten zu können und Methoden zu testen, welche das jeweili-

ge Verhalten fairer gestalten. Bei dem eben skizzierten naiven Verfahren ist dies nicht möglich. Jeder Knoten übernimmt gleichviel Arbeit, egal wie groß sein Interesse ist [8]. Durch Churn entstandene Arbeit kann nicht fair verteilt werden, da kein Mechanismus vorgesehen ist hierbei die Größen der Topics, wie in Kapitel vier dargestellt, zu verändern. Es erfolgte eine Dreiteilung des Systems. Von nun an wird in dieser Arbeit von drei verschiedenen Arten an virtuellen Knoten gesprochen, den *Pubnodes*, *Workingnodes* und *Churnnodes*. Diese Knoten sind in drei verschiedenen Strukturen angeordnet. Die Trennung in Pub- und Workingnode wurde durchgeführt, um Arbeit verteilen zu können. Das heißt, dass Workingnodes Arbeit zum Erhalt des Systems übernehmen, während Pubnodes vom System profitieren. Zusätzliche Churnnodes übernehmen ebenfalls Arbeit, sollten allerdings die faire Lastverteilung nicht stören und werden deshalb gesondert behandelt.

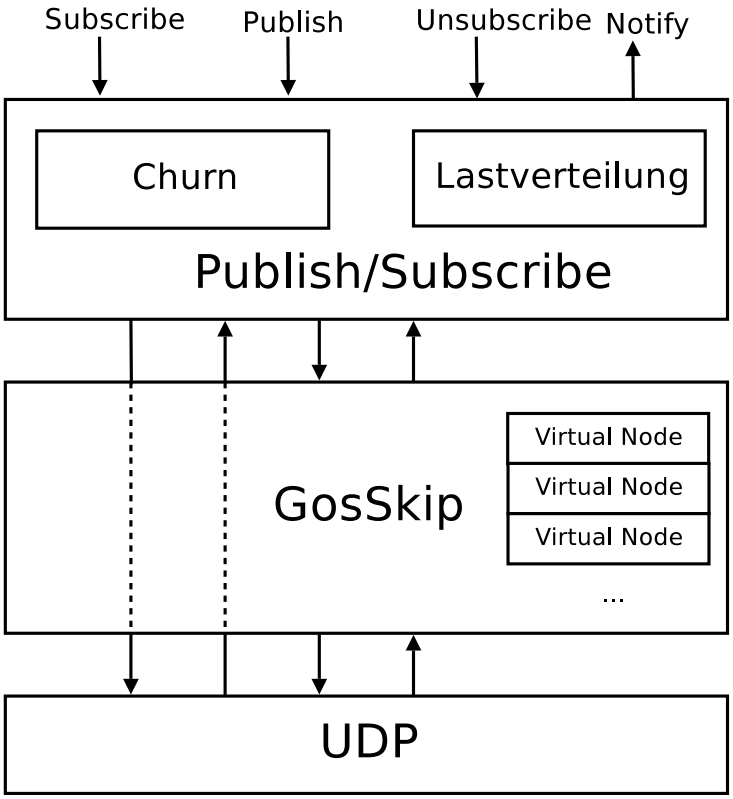

Abbildung 5.1: Schematische Darstellung des Schichtenmodells dieses dynamischen pub/sub-Systems

Pubnodes sind dazu da, um zu einem Topic zu Subscriben, dort alle interessanten Ereignisse zu sammeln und Publishen zu können. Diese Knoten wurden speziell dazu konzipiert, keine Arbeit zum Erhalt der Struktur des

Systems beizutragen. Außer beim Publishen, bei dem sie Ereignisse speichern und weiterleiten müssen, sind diese Knoten zumeist passiv. Strukturiert angeordnet, bilden sie einen simplen, doppelt verketteten Ring, den Pubnodering, über den Publish-Nachrichten geflutet werden können.

Diese passiven Knoten werden von virtuellen Knoten, welche nicht zwangsläufig auf dem selben physikalischen Knoten zu finden sind, verwaltet. Verwalterknoten haben den Namen *Workingnode* erhalten, da sie aktiv Arbeit übernehmen. Sie werden in einem GosSkip-Ring angeordnet, müssen dort Gossip-Nachrichten versenden, allgemein Informationen weiterleiten, entscheiden, ob ein Nachbarknoten ausgefallen ist und dafür sorgen, dass von ihnen verwaltete Knoten mit entsprechenden Informationen versorgt werden.

Zur kurzzeitigen Behebung von unfairem Verhalten bezüglich Churn, wurden weitere virtuelle Knoten erdacht, die in einem zweiten GosSkip-Ring, zusammen mit den Workingnodes, angeordnet werden. Diese sogenannte *Churnnodes* sorgen dafür, dass der Bereich eines Topics vergrößert und früher getroffen wird. Arbeit, welche durch Weiterleitungen von lookup- und join-Nachrichten anfällt, soll dadurch schneller in das Gebiet des Topics verlagert werden.

Der Aufwand, Gossip-Nachrichten zu verteilen, wurde durch die Aufteilung in einen Ring aus Workingnodes, den *Workingnodering*, und einen Ring aus Working- und ChurnNodes, den *Churnring*, mehr als verdoppelt! Der Mehraufwand an Netzwerklast, der durch die Aufteilung des Systems in Pub- und Workingnodes ensteht, ist mit zusätzlichen regelmäßigen Alive-Nachrichten und doppelter Arbeit durch lookup (einmal um den Pubnode einzufügen und ein zweites mal um einen, den Pubnode verwaltenden Workingnode einzufügen) sehr grob zu beaufschlagen. Ob sich die Mehrarbeit lohnt, wird im späteren Verlauf diskutiert.

Die Funktionalität ist auf zwei Schichten verteilt (Abb. 5.1). Zum einen eine Schicht, auf der das GosSkip-Overlay und Funktionalitäten dahingehend zu finden sind und zum anderen eine Schicht, auf der das pub/sub mit Funktionalität zu finden ist. Theoretisch kann man über eine darüberliegende Schicht mittels einer Schnittstelle Subscribe(), Unsubscribe(), Publish(), Notification() das pub/sub-System transparent verwenden. Optionale Module zur Behandlung von Lastverteilung und Churn sind auf pub/sub-Ebene eingefügt. Jede Schicht muss eine Menge an virtuellen Knoten vorhalten und verwalten können.

Die GosSkip-Ringe befinden sich logisch gesehen auf der GosSkip-Overlay Ebene, der Pubnodering dagegen auf der pub/sub-Ebene. Verwaltungsaufgaben der Workingnodes werden auf pub/sub-Ebene durchgeführt, während strukturaufbauende und -erhaltende Aufgaben dagegen auf GosSkip-Overlay-Ebene durchgeführt werden.

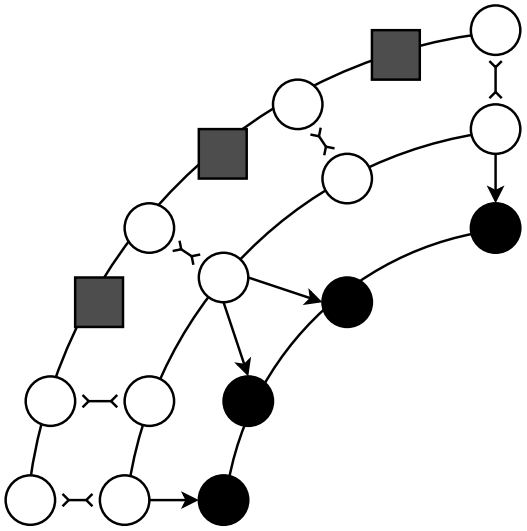

Abbildung 5.2: Schematische Darstellung des dynamischen pub/sub System

Eine schematische Struktur ist in Abbildung 5.2 zu erkennen. Es ist ein Ausschnitt, welcher die drei Ringe präsentiert. Schwarze Knoten stellen die im *Pubnodering* angeordneten *Pubnodes* dar. Die grauen Rechtecke symbolisieren *Churnnodes*, welche ausschließlich im *Churnring* zu finden sind. Die weißen Knoten stellen Workingnodes dar. Zwischen diesen sind Verbindungselemente eingezeichnet. Da Workingnodes auf pub/sub-Schicht transparent als jeweils ein Knoten gesehen werden, allerdings auf GosSkip-Schicht in zwei Ringe aufgeteilt sind, wird mit Hilfe des Verbindungselements dieser Zusammenhalt symbolisiert. Lange Verbindungen der GosSkip-Ringe wurden der Übersichtlichkeit wegen weggelassen, bestehen allerdings in beiden GosSkip-Ringen.

Die Kommunikation innerhalb des Systems läuft über Nachrichtenaustausch. Das verwendete Transportprotokoll war hierbei UDP. Über das Verhalten des Systems mit Nachrichtenausfällen werden in der Arbeit keine Aussagen gemacht, da von verfügbaren und zuverlässigen Kanälen ausgegangen wurde. Einzig und allein Knotenausfälle wurden berücksichtigt.

5.2 GosSkip-Overlay

Die ringgleiche Overlay Struktur von GosSkip soll, wie vorhergehend erwähnt, als Anordnung der virtuellen Knoten dienen. Der Name, anhand dem diese angeordnet werden, ist eine zweigeteilte Identifikation mit einer entsprechenden Ordnungsrelation. Um seine Position im System zu finden, muss

ein Knoten eine join-Nachricht senden, welche genau so wie eine lookup-Nachricht durchs System weitergeleitet wird. Beim Einfügen an der richtigen Stelle bekommt ein Knoten Informationen über seinen Vorgänger und seinen Nachfolger. Von nun an werden diese als linker und rechter Nachbar auf Ebene 0 bezeichnet. Ab dem Zeitpunkt des Einfügens werden in periodischen Abständen Gossip-Nachrichten an den rechten Nachbarn auf Ebene 0 propagiert. Mit Hilfe von Gossip-Nachrichten werden die Nachbarschaftslisten nun nach und nach bis jeweils zur maximalen Höhe $\log(N)$, sei N die Anzahl der Knoten im System, aufgebaut. Diese maximale Höhe kann dynamisch anhand der Systemgröße adaptiert werden. Ein, ebenfalls über die Systemgröße angepasster, Sicherheitsmechanismus stellt mit Hilfe einer Reparaturliste eine k-Fehlertoleranz zur Verfügung. Die Knoten dieser Reparaturliste werden bei Ausfällen der Reihe nach angefragt, bis ein aktiver Knoten gefunden wird. Die Liste wird ebenfalls verwendet, sobald ein Knoten das System verlässt, da dieser dazu einfach das Senden von Gossip-Nachrichten einstellt, was dem Verhalten bei einem Ausfall entspricht. Es werden also keine speziellen Benachrichtungen beim Verlassen versendet.

5.2.1 Zweigeteilte Identifikation

Unter der Annahme, ein physikalischer Knoten tritt nur einmal zu einem Topic bei, reicht es, das Schlüsselwort eines Topics als Identifikation des virtuellen Knotens zu nutzen. Zusammen mit einer Identifikation des physikalischen Knotens, beispielsweise der ip-Adresse, ist der virtuelle Knoten eindeutig im pub/sub- und P2P-System zu identifizieren. Wenn nun ein physikalischer Knoten zweimal dem gleichen Topic beitritt, oder wie in dem implementierten System, mehrere virtuelle Knoten im gleichen Topic arbeiten müssen, so muss man diese Knoten unterscheiden können.

Um die Eindeutigkeit eines Knotens zu gewährleisten wurden deshalb drei Merkmale herangezogen. Zum ersten einen sha1-Hash über dem Schlüsselwort des Topics, zum zweiten einen zufälligen sha1-Hashwert, welcher eine Ordnung innerhalb eines Topics impliziert und zu letzt die ip-Adresse des physikalischen Knotens. Im weiteren Verlauf der Arbeit wird über den Hashwert des Schlüsselwortes als Topic und den zufälligen Hashwert als Topicorder gesprochen. Topic und Topicorder werden zusammen als Schlüsselpaar bezeichnet. Diese Paare werden nun als 2-Tupel (x,y) geschrieben, wobei das erste Element x, das Topic und das zweite Element y, die Topicorder bezeichnet. Die Eindeutigkeit basiert auf der hohen Wahrscheinlichkeit, dass Hashwerte nicht doppelt verwendet werden.

Eine Ordnungsrelation $<$ kann für die Menge der Schlüsselpaare M definiert werden $< \subseteq M \times M$. Die grobe Ordnung kann anhand des Topics

festgemacht werden: $Topic_i < Topic_{i+1}$, seien i und i+1 $\in$ der Menge der sha1-Hashwerte, $Topic_i$ der direkte Vorgänger von $Topic_{i+1}$ und i = 0 das Infimum, sowie i = max das Supremum. Die Topicorder emöglich eine feingranulare Ordnung, sobald $Topic_j = Topic_k$, wobei k,j $\in$ der Menge der sha1-Hashwerte: $Topicorder_i < Topicorder_{i+1}$, seien i und i+1 $\in$ der Menge der sha1-Hashwerte, $Topicorder_i$ der direkte Vorgänger von $Topicorder_{i+1}$ und i = 0 das Infimum, sowie i = max das Supremum. Max beläuft sich standardmäßig auf 2^{160} in der verwendeten sha1 Implementierung. Beispiele hierfür sind $(1, 2) < (2, 1)$ und $(5, 8) < (10, 9)$.

Mit Hilfe dieser Ordnung kann man nun bestimmen, ob eine Identifikation zwischen zwei anderen liegt. Diese Festlegung ist in Ringen nicht trivial, da Entscheidungen über die Ringgrenze, also das Gebiet, bei dem der Knoten mit der minimalen Identifikation und der maximalen Identifikation aufeinander treffen, hinweg getroffen werden müssen. Der Knoten, mit dem Schlüsselpaar (5,6), ist nicht zwischen den Knoten mit den Schlüsselpaaren (6,1) und (1,2) einzuordnen. Dahingegen ist er aber sowohl zwischen den Knoten mit den Schlüsselpaaren (5,1) und (6,1), als auch zwischen den Knoten mit den Schlüsselpaaren (4,3) und (1,2) einzuordnen.

Diese Schlüsselpaare bieten einen Namen um in GosSkip-Ringen eine Sortierung zu finden. Dieses Verhalten ist in Abbildung 5.3 visualisiert. Zusammen mit der ip-Adresse eines physikalischen Knotens sind sie eindeutig und deshalb als Nachbarschaftseinträge geeignet. Ein DHT -ähnlicher Ansatz kann hieraus verwendet werden. Ein virtueller Knoten übernimmt die Verantwortung für Schlüsselpaare die zwischen seinem eigenen Paar und dem seines rechten Nachbarn liegen.

5.2.2 Strukturaufbau

Die Overlay-Struktur ist mit all ihren Verbindungen auf allen Ebenen nicht, wie aus magischer Hand, sofort nach dem Beitritt der Knoten in das Overlay, aufgebaut. Stattdessen wird es in Anlehnung an den in [2] vorgestellten Algorithmus konstruiert. Ein virtueller Knoten kennt initial nur seinen direkten Vorgänger und Nachfolger, auch rechter und linker Nachbar auf Ebene 0 genannt. Diese werden in zwei verschiedenen Nachbarschaftslisten gehalten, der rechten und der linken. Beide Nachbarschaftslisten werden nun gleichzeitig aufgebaut. Auf Ebene i $\in \mathbb{N}$ enthalten sie Einträge von Nachbarn, die jeweils eine Skiplänge k $\in \mathbb{N} > 0$ an Verbindungen der Ebene i-1 überspringen. In der verwendeten Implementierung wurde k auf zwei eingestellt und deshalb wird im Folgenden in Beispielen hauptsächlich auf diese Parametrierung eingegangen.

Ein neu in die Struktur integrierter Knoten sendet periodisch nach rechts

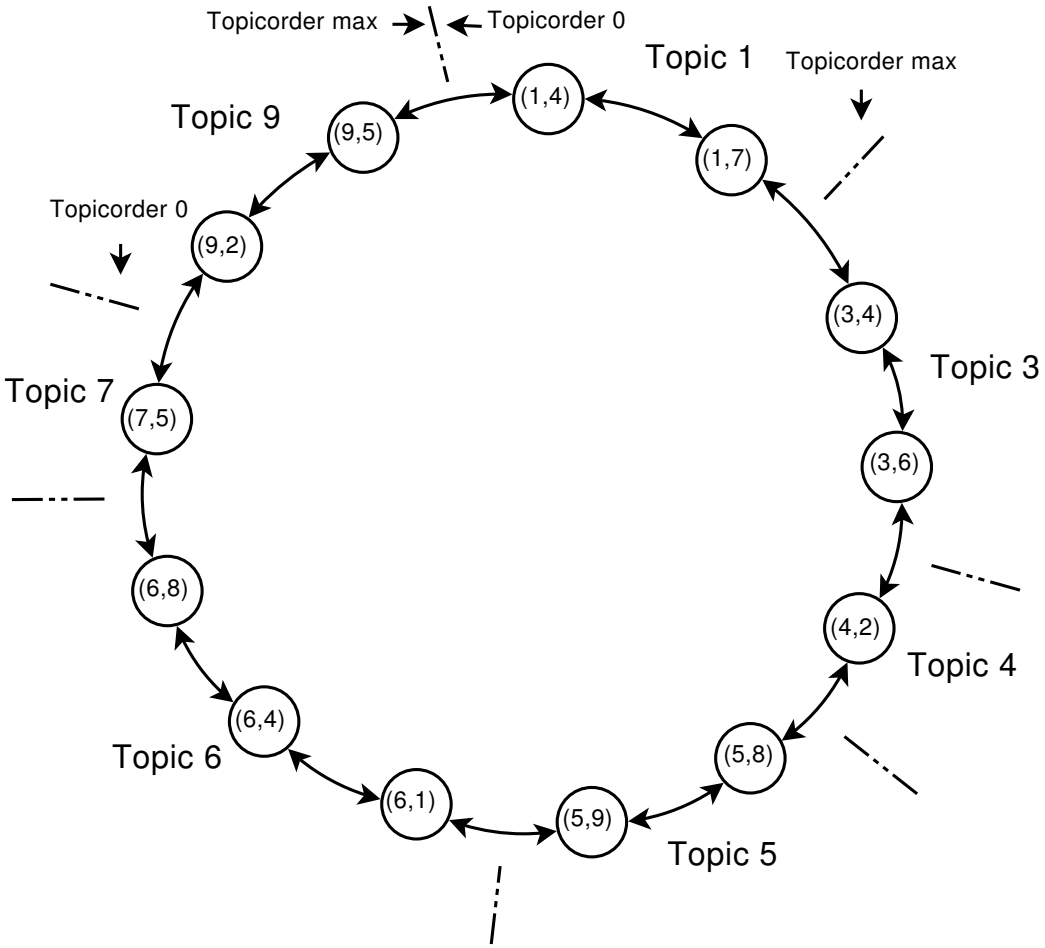

Abbildung 5.3: Aufteilung des Gebietes durch die Schlüsselvergabe

auf Ebene 0 und später auf allen Ebenen sogenannte Gossip-Nachrichten. In diesen Gossip-Nachrichten wird eine Menge an Tripeln (oder auch Gossip-Einträge genannt) transportiert. Einjedes dieser Tripel besteht aus einer Identifikation eines physikalischen Knotens, einem Namen und einem counter. In dieser Arbeit entspricht dies der ip-Adresse, einem Schlüsselpaar sowie einem counter, welcher bei 0 startet und bei jedem hop inkrementiert wird. Einjeder Knoten erhält Gossip-Nachrichten-Tripel mit $counter < k$, speichert diese zwischen und fügt ein Tripel mit Informationen über sich selbst vor dem Weitersenden hinzu.

Erhält ein Knoten auf Ebene 0 eine Gossip-Nachricht, so fügt er die wichtigsten Informationen des Tripels mit $counter = k - 1$ aus der Nachricht auf Ebene 1 in seine linke Nachbarschaftstabelle ein, bestehende veraltete Daten werden dabei überschrieben. Damit eine Duplex Verbindung entste-

hen kann und da per se nicht genug Informationen, um diese weiter nach rechts zu versenden, vorhanden sind, wird auf Ebene 1 (und ebenso auf allen anderen Ebenen mit ungerader Nummer) mit dem periodischen Senden der Gossip-Nachrichten nicht nach rechts, sondern nach links fortgefahren. Empfängt also ein Knoten auf Ebene 1 die Nachricht, so kann ein Knoten die wichtigsten Daten aus der Nachricht entsprechend verarbeiten. Zum einen müssen Informationen des Tripels mit $counter = 0$ auf Ebene 1 in die rechte Nachbarschaftsliste eingetragen werden, denn dies ist der Eintrag seines direkten linken Nachbarn auf Ebene 1, zum andern kann er den Tripel mit $counter = k - 1$ auswerten und als linken Nachbarn auf Ebene 2 verwenden. Diese Vorgehensweise wird bis zur, dynamisch ermittelten, maximalen Ebene des Systems iteriert. Die geraden Ebenen werden einfach anhand des Terms (Ebene mod 2) = 0 von den ungeraden unterschieden.

Anders als im ursprünglichen Algorithmus werden, sobald Gossip-Nachrichten auf Ebene 0 und mit counter 0 eingetroffen sind, Informationen für die darüberliegende pub/sub-Schicht aus den Nachrichten herausgeholt und spezielle Methoden angewandt, um die Reparaturliste zu verwalten.

In Anbetracht von fehlerhaften Nachbarn, welche ohne explizite Nachricht das System verlassen, kommt hier ein softstate-Ansatz zum tragen. Alle Einträge der Nachbarschaftsliste werden mit Timeouts versehen. Laufen diese ab, so vermutet ein Knoten, dass ein entsprechender Nachbar auf Ebene i $\in$ $\mathbb{N}$ ausgefallen ist. Da auf geraden Ebenen Gossip-Nachrichten nach rechts gesendet werden, kann durch Ausbleiben derer in linken Nachbarschaftslisten der Eintrag gelöscht werden, auf ungeraden Ebenen umgekehrt. Um auch über die Gegenrichtung Aussagen bezüglich Ausfällen machen zu können, antwortet ein Knoten, welcher Gossip-Nachrichten erhält, mit einer Antwort-Nachricht. Nach Ausbleiben der Antwort-Nachrichten können entsprechende Einträge aus der Nachbarschaftsliste gelöscht werden. Einzig und allein auf Ebene 0 müssen Aktionen ergriffen werden, um die Struktur aktiv zu reparieren. Auf allen anderen Ebenen wartet man auf die nächsten Gossip-Nachrichten. In Anbetracht des Zwei-Armeen Problems, ist auch dies nur eine auf Vermutungen basierte Methode, welche allerdings selbst bei einer fehlerhaften Vermutung kein Problem darstellt. Wie in einem späteren Abschnitt noch dargelegt arbeitet in einem solch unglücklichen Fall, die sogenannte *Linknotification* zur Ringreparatur.

Diskussionswürdig sind die Antwort-Nachrichten, da sie zusätzliche Netzwerklast produzieren. Nach endlicher Zeit ersetzt man durch Informationen über einen aktiven Knoten den nicht mehr aktiven Eintrag auf Ebene i $\in$ $\mathbb{N}^+$. Andererseits ist zu bedenken, dass es lange dauert, bis auf den hohen Ebenen die neuen Einträge notiert werden und dementsprechend lang alte Informationen verwendet werden. Die Zeit, die benötigt wird um einen falschen

Eintrag auf Ebene i zu aktualisieren, variiert in Abhängigkeit von i. Da die verteilte Suche darauf beruht, mit hoher Wahrscheinlichkeit aktive Knoten als Nachbarn anzutreffen und vorallem auf den hohen Ebenen mit der Suche begonnen wird, ist bei diesem trade-off die Wahl auf Antwort-Nachrichten gefallen, welche über einen Parameter gesteuert werden können.

5.2.3 Dynamische Größe der Nachbarschaftslisten

GosSkip bietet durch die Eigenschaften, die ähnlich einer Perfekten Skip Liste sind, die Möglichkeit, die Anzahl der langen Verbindungen und damit die Größe der Nachbarschaftslisten anhand der Identifikationen der Nachbarn zu adaptieren. Zusätzlich erhält man die Möglichkeit, die Anzahl der beteiligten virtuellen Knoten des Systems abzuschätzen.

Nach den folgenden Regeln wurde die Nachbarschaftsliste dynamisch aufgebaut. Sei N $\in \mathbb{N}$ die Anzahl der virtuellen Knoten, max $\in \mathbb{N}$ die oberste Ebene, counter $\in \mathbb{N}$ der in einer Gossip-Nachricht mitpropagierte Zähler, k $\in \mathbb{N}$ die Skiplänge, m $\in \mathbb{N}$ eine beliebige Anzahl an Ebenen:

1. Verbindungen auf Ebene 0 existieren bei jeder Systemgröße.

2. Die Größe der rechten Nachbarschaftstabelle wird um eins erhöht, wenn ein Knoten G Gossip-Nachrichten auf Ebene max+1 erhält und die Identifikation des rechten Nachbarn H auf Ebene max zwischen der Identifikation von G und der Identifikation des Gossip-Eintrags I mit $counter = k - 1$ liegt.

3. Die Größe der linken Nachbarschaftstabelle wird um eins erhöht, wenn ein Knoten G Gossip-Nachrichten auf Ebene max+1 erhält und die Identifikation des linken Nachbarn H auf Ebene max zwischen der Identifikation des Gossip-Eintrags I mit $counter = k - 1$ und der Identifikation von G liegt.

4. Die Größe der rechten Nachbarschaftstabelle wird um m verringert, wenn ein Knoten G Gossip-Nachrichten auf Ebene i erhält und die Identifikation des Gossip-Eintrags I mit $counter = k - 1$ zwischen der Identifikation von G und der Identifikation des rechten Nachbarn H auf Ebene i-1 liegt. Es ergibt sich m zu Größe der rechten Nachbarschaftsliste - i.

5. Die Größe der linken Nachbarschaftstabelle wird um m verringert, wenn ein Knoten G Gossip-Nachrichten auf Ebene i erhält und die Identifikation des Gossip-Eintrags I mit $counter = k - 1$ zwischen der Identifi-

kation des linken Nachbarn H auf Ebene i-1 und der Identifikation von G liegt. Es ergibt sich m zu Größe der linken Nachbarschaftsliste - i.

Befolgt man die Regeln, so kann ein Knoten mit hoher Wahrscheinlichkeit davon ausgehen, dass die Anzahl der Teilnehmer des Systems zwischen 2^{max} und 2^{max+1} beschränkt ist. Bei mehr Teilnehmern, wäre eine neue lange Verbindung generiert worden, bei weniger wären lange Verbindungen abgebaut worden. Durch die Reaktionszeit des Systems, also bis ein Knoten von seinem 2^{max+1}-ten Nachbarn informiert wird, trifft er noch die falsche Annahme über eine Systemgröße abweichend 2^{max+1} bis 2^{max+2}.

Dementsprechend kann jeder Peer auch einschätzen, wie groß ein Bereich ist. In diesem speziellen Fall soll bestimmt werden, wieviele Prozesse für ein Topic arbeiten müssen. Jeder Knoten schaut jeweils in seiner rechten und linken Nachbarschaftsliste nach dem höchsten Eintrag, dessen Schlüsselpaar im ersten Element noch mit dem Schlüsselpaar des Knotens übereinstimmt. Um zu vermeiden, dass eine lange Verbindung verwendet wird, welche durch zu weites überspringen wieder am anderen Ende des Bereiches eintrifft, kann der zweite Eintrag des Schlüsselpaares verwendet werden. Das heißt, ein Eintrag, welcher zur Abschätzung nach rechts verwendet wird, muss immer eine Topicorder besitzen, die größer ist, als die des abschätzenden Knotens. Nach links ist dies genau umgekehrt. Seien r, l $\in \mathbb{N}$ die Ebene dieser Einträge, so lässt sich das Gebiet mit $k^r + k^l$ grob abschätzen. Dabei ist ein Fehler $\varepsilon \in \{f \mid f \in \mathbb{N} \wedge 0 \leq f \leq (k^{r+1} - k^r) + \left(k^{l+1} - k^l\right) - 2\}$ zu erwarten, da ein Knoten keine Aussagen über alle Nachfolger zwischen k^r und k^{r+1} bzw. k^l und k^{l+1} treffen kann.

Die Einschätzung verbessert sich, wenn man mit Hilfe der Gossip-Nachrichten, die zwischen diesen Nachbarn ausgetauscht werden, schon Einschätzungen über ihre Entfernung zum rechten bzw. linken Rand mitschickt. Ein Knoten, welcher nach rechts Gossip-Nachrichten oder Antworten sendet, kann mit dem Wissen aus seiner lokalen Sicht eine bessere Abschätzung treffen und diese mitpropagieren lassen. Mit Hilfe einer langen Verbindung, die auf den k^i-ten, sei i $\in \mathbb{N}$, Nachfolger zeigt, und mit dessen Einschätzung über die Entfernung zum rechten bzw. linken Rand des Gebietes eines Topic, welche mitpropagiert wurde, kann ein Knoten seine lokale Einschätzung berechnen. In einem GosSkip-Ring ohne Churn konvergiert jede grobe Einschätzung zu einer perfekten Einschätzung. Durch Churn im GosSkip-Ring erhält man einen Fehler.

5.2.4 Routing-Aspekte im GosSkip-Ring

Das Routing wird unter drei Aspekten betrachtet. Zum einen müssen virtuelle Knoten ihren Platz im Overlay finden. Dazu werden join-Nachrichten innerhalb des Systems vermittelt. Zum anderen müssen Verwalter für bestimmte Schlüsselpaare gefunden werden können. Dazu werden lookup-Nachrichten innerhalb des Systems vermittelt. Zuletzt müssen noch virtuelle Knoten gefunden werden, die den k-ten Nachbarn im GosSkip-Ring auf Ebene 0 darstellen. Dazu wurden spezielle lookup-Nachrichten innerhalb des Systems vermittelt. Die Art und Weise, wie join und lookup durch das System propagiert werden unterscheidet sich in keiner Weise, jedoch die Reaktion bei den Zielknoten unterscheidet sich gewaltig. Die Suche nach dem k-ten Nachbarn unterscheidet sich leicht von den normalen join- und lookup-Protokollen.

Da Nachbarschaftsbeziehungen den Verbindungen einer Skip-Liste ähneln, kann man, genauso wie man sich innerhalb einer Skip-Liste an den Verbindungen entlanghangelt, die Nachricht von Knoten zu Knoten propagieren lassen. Vom sogenannten Bootstraporacle, einem Dienst der Informationen über Knoten zurückliefert, bekommt einjeder, der eine Nachricht innerhalb des Systems vermitteln will, einen zufälligen Knoten. Ein physikalischer Knoten könnte, falls er einen virtuellen Knoten im System stellt, auch diesen als Start des lookups oder joins verwenden. Initial wird die lookup- oder join-Anfrage also an einen beliebigen Knoten gesendet und von dort aus nach rechts weitergeleitet. Man startet damit, das gesuchte Schlüsselpaar C (entweder eines join-Gesuches oder eines lookup-Gesuches) mit dem Schlüsselpaar des Knotens A, der die Nachricht weiterleiten soll und des Schlüsselpaars B, eines auf höchster Ebene in der rechten Nachbarschaftsliste eingetragenen Knotens, zu vergleichen. Ist der Eintrag C zwischen A und B einzusortieren, so wird die Suche eine Ebene weiter unten fortgesetzt. Ist der Eintrag nicht zwischen A und B einzusortieren, so muss das Gesuch an den nächsten Nachbarn auf dieser Ebene gesendet werden, um dort die Suche fortzuführen. Ist man durch Iteration der Vorschriften bei einem Knoten auf Ebene 0 an sein Ziel angelangt so kann dieser entsprechende Reaktionen ausführen. Entweder die geforderten Informationen über sich selbst zurückliefern oder durch Benachrichtigung des join-Anfragenden und des momentanen rechten Nachbarn auf Ebene 0 diesen Knoten einzugliedern.

Um diese Art des Routings noch weiter zu beschleunigen, ist es möglich, Wissen innerhalb des physikalischen Knotens über alle seine virtuellen Knoten im GosSkip-Ring auszunutzen. Lokal kann man entscheiden, welcher seiner Knoten am nähesten am Ziel dran ist und von dort aus die Nachricht weiterleiten. Hier wird deutlich, dass, sobald mehrere virtuelle Knoten im System vorhanden sind, eine gewisse globale Verbesserung erreicht wird, da

sich die Anzahl der hops verringert (im Detail in [2] zu sehen).

Sei k $\in \mathbb{N}$, so kann man den k-ten Nachbarn im Ring finden, indem man die Suche direkt bei dem Knoten startet, dessen Nachbarn man erhalten möchte und die Eigenschaften der Perfekten Skip-Liste ausnutzt. Für die folgenden Annahmen wurde die Skiplänge, der variable Systemparameter, welcher bestimmt wie weit lange Verbindungen reichen, mit zwei angenommen. Das ganze Prinzip ist mit jegliche Skiplänge verwendbar. Die erdachte Suche basiert darauf, dass die Nachbarschaftslisten auf den Ebenen i $\in \mathbb{N}$ exakt den 2^i-ten Nachbarn beinhalten. Man beginnt auf oberster Ebene mit der Suche und vergleicht immer den mitpropagierten counter k $\in \mathbb{N}$ und 2^i. So sendet man eine Nachricht zum erstbesten Knoten in der Nachbarschaftsliste, der die Ungleichung $2^j \leq k$, sei j $\in \mathbb{N}$, erfüllt. Natürlich sollte davor k durch $k-2^j$ angepasst werden. Das Abbruchkriterium ist offensichtlich $k = 0$. Durch noch nicht korrigierte Nachbarschaftslisten, nach Ausfällen oder Eingliederungen, kann es hierbei zu einem Fehler $\varepsilon \in \mathbb{Z}$ kommen und man erhält nur den k + ε-ten Nachbarn.

5.2.5 Reparaturlisten und Fehlertoleranz

Wie schon zuvor erwähnt, verlässt ein Knoten das System indem er einfach keine Gossip-Nachrichten mehr versendet. Das gleiche Verhalten zeigt sich bei einem Ausfall. Die Overlay-Topologie muss, nachdem eine dieser zwei Arten von Ausfällen aufgetreten ist, wieder repariert werden, um das Arbeiten mit dem P2P-System auch in Zukunft zu ermöglichen.

Der einfachste Ansatz wäre es, Informationen über Nachbarn, die schon durch Gossip-Nachrichten auf Ebene 0 durch das System propagiert werden, in einer Reparaturliste zu speichern. Man erhielte Informationen über Knoten, die links von einem liegen. Das Problem hierbei ist zum einen, dass Informationen bei einer Konfiguration mit Skiplänge $k = 2$, also immer wenn 2^0, 2^1 ... Nachbarn übersprungen werden, auf jeder Ebene nur 2 hops weit getragen werden. Den direkten Nachbarn, über dessen Ausfall man Vermutungen anstellt, sollte man nicht in einer Reparaturliste halten, denn damit wäre die Liste mit nur $k - 1$ Knoten gefüllt. Dies würde im Falle einer Parametrisierung mit $k = 2$ zu einer Liste mit nur einem Eintrag führen. Für große Systeme ist damit aber die Fehlertoleranz bei Knotenausfällen zu gering. Die Größe der Reparaturliste sollte möglichst adaptiv mit der Größe des Systems skalieren.

Die Lösung dieses Problems liegt darin, diese Informationen weiter propagieren zu lassen als geplant. Adaptiv anhand der Systemgröße, also über die Höhe der Nachbarschaftslisten, könnte ein jeder Knoten autonom entscheiden, ab welchem counter er einen Eintrag mit den Gossip-Nachrichten

nicht mehr weiter versendet. Dieser Wert wird fortan als *threshold* bezeichnet, und wurde immer mit der Höhe der Nachbarschaftsliste gleichgehalten. Da bei jeder Systemgröße eine gewisse minimale Ausfallsicherheit gewährleistet werden sollte, gibt es die Möglichkeit einen minimalen *threshold* anzugeben.

Da während der Zeit, in der der GosSkip-Ring kaputt ist, neue Peers dem System beitreten können und diese für gewöhnlich von der linken Seite eingefügt werden, entstehen möglicherweise Entartungen des Ringes nach einer Reparatur. Da Gossip auf Ebene 0 bekanntlich nach rechts gesendet wird, wird der Knoten rechts des Ausgefallenen zur Ringreparatur herangezogen. Nachdem Informationen über neue Knoten nach dem Ausfall eines Knotens nicht über diesen hinweg getragen werden, hält derjenige, der reparieren soll, nicht unbedingt die aktuellsten Informationen. Sind nun neue Knoten zwischen dem Ausgefallenen und dessen ehemaligen linken hinzu gekommen, so entstehen unschöne Verästelungen. Das Dilemma wäre nicht so groß, wenn der Knoten links von dem Ausgefallenen reparieren könnte. So ist es eine designtechnische Entscheidung gewesen Reparaturlisten, auf Kosten der Netzwerklast, immer mit den Gossip-Antworten und nicht mit den Gossip-Nachrichten selbst nach links mitpropagieren zu lassen. Ein neu gejointer Knoten kann nun auch die Reparaturliste desjenigen übernehmen, welcher ihn eingefügt hat, während dieser den ersten Eintrag in der Reparaturliste durch den neu eingefügten Knoten ersetzen kann. Die Reparatur geschieht dann, indem Gossip-Nachrichen auf Ebene 0 an den, aus der Reparaturliste entnommenen, neuen rechten Nachbarn geschickt werden.

Das P2P-System ist weder nach *threshold+1* aufeinander folgenden Ausfällen zwangsläufig kaputt, noch wenn kein Eintrag in der Reparaturlist zu finden ist. Mit einer gewissen Wahrscheinlichkeit existieren auf einer Ebene i $\in \mathbb{N}$ mit $2^i > threshold$ noch aktive Nachbarn. Diese noch aktiven Nachbarn könnte man nun zur Ringreparatur heranziehen. Es ist sinnvoll Nachbarn aus der Nachbarschaftsliste zu wählen, welche lange Verbindungen in die Richtung beinhaltet in welche man repariert. Diese sind näher am eigentlichen neuen Nachbarn auf Ebene 0.

Die zweite Möglichkeit einen entarteten Ring zu erhalten ist, wenn zur Reparatur des Ringes Nachbarn auf hohen Ebenen angefragt werden müssen. In 5.4 ist dargestellt, wie dieser Fall entsteht, wenn ein Knoten keine Reparaturliste besitzt. Es kann geschehen, dass nun von zwei oder mehr verschiedenen Knoten Gossip-Nachrichten bei einem Knoten zusammentreffen. Bei zwei verschiedenen Knoten, die an einen dritten eine Gossip-Nachricht auf Ebene 0 schicken, ist mit gewisser Wahrscheinlichkeit hinter einem der Knoten eine Kette an Knoten und hinter dem anderen der Ring. Beispielsweise ist dies bei Knoten (7,5) in Abbildung 5.4 durch die Knoten (3,7) und (6,9) der Fall. Der, durch konkurrierende Gossip-Nachrichten belastete

Knoten (7,5), ist mit diesen Informationen nun in der Lage zu entscheiden, welchen der beiden Knoten er eigentlich als Vorgänger bevorzugen würde. Dem zweiten, nicht bevorzugten Knoten, schickt er eine Nachricht mit Informationen über den bevorzugten Knoten, denn dieser ist dessen neuer Nachfolger. Da nach einer solchen Nachricht, einer sogenannten *Linknotification*, sofort eine neue Gossip-Nachricht versendet wird und der Entscheidungsprozess möglicherweise erneut stattfindet, konvergiert der Ring relativ schnell zurück zum Ring auf Ebene 0.

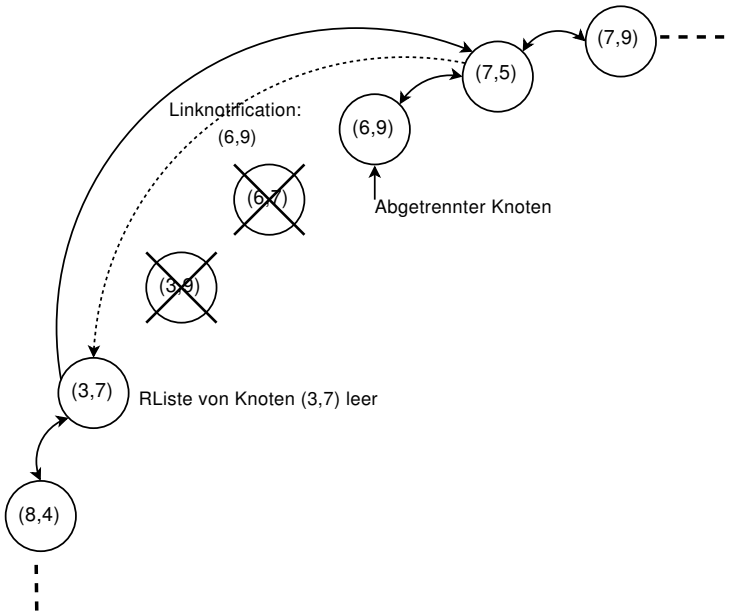

Abbildung 5.4: Exemplarische Darstellung wie der Ring auch ohne Reparaturliste repariert werden kann

Bei einem Beitritt ins System hat natürlich jeder Knoten noch eine leere Reparaturliste. Diese kann sofort befüllt werden, indem der neue Knoten die Reparaturliste dessen übernimmt, der ihn eingefügt hat. Der Knoten, welcher ihn eingefügt hat, weiß, dass der erste Knoten seiner neuen Reparaturliste sein alter rechter Nachbar ist. Er hat den neue Knoten ja zwischen sich und diesen ins System eingefügt.

Paralelle Beitrittsgesuche stellen kein Problem dar, da diese seriell am Knoten abgearbeitet werden.

5.3 Konzepte der Pub/Sub-Schicht

Die pub/sub-Ebene ist auf den GosSkip-Ring aufgesetzt und muss dafür sorgen, dass Subscriptions, Unsubscriptions, Publications, als auch Benachrichtigungen für eine darüberliegende Applikation transparent verarbeitet werden. Subscriptions müssen entgegengenommen werden und die Möglichkeit bieten, alle interessanten Ereignisse auszuliefern. Dazu müssen Pubnodes, Knoten, die für das Publishen und Benachrichtigen zuständig sind, erzeugt werden und entsprechende Subscription-Gesuche durch das System zu den passenden Workingnodes, Knoten, die für das Aufrechterhalten der Struktur verantwortlich sind, geleitet werden. Bei Bedarf sollten Churnnodes, die zur faireren Verteilung der Arbeit bei hohem Churn gedacht sind, zusätzlich Arbeit übernehmen. Sowohl Workingnodes, als auch Churnnodes werden mit genügend Informationen versehen, um in zwei GosSkip-Ringen angeordnet zu werden. Zum einen existiert ein Churnring, in dem Workingnodes und Churnnodes koexistieren. In diesem Ring werden alle join- und lookup-Gesuche geroutet. Spezielle Vorkehrungen waren hier für Churnnodes vorzusehen, da diese beim lookup nicht als Ziel einer Subscription getroffen werden sollten. Der zweite Ring, der Workingnodering, fügt alle Workingnodes in einem getrennten Ring zusammen. Hier wird die Möglichkeit geboten, relativ gleichverteilt über alle existierenden Workingnodes, durch Verwendung von lookupx und einer Abschätzung der Größe des Ringes, Anfragen zu stellen. Workingnodes verwalten Pubnodes. Jeder Workingnode wird durch ein join in die GosSkip-Ringe für einen bestimmten Bereich an Schlüsselpaaren verantwortlich gemacht. Ein Subscription-Gesuch muss deshalb zum Workingnode mit einer passenden zweigeteilten Identifikation, also einem passenden Schlüsselpaar, gelangen. Durch periodische Alive-Nachrichten und passende Antworten überprüfen sie gegenseitig, ob der jeweilige Gegenüber noch aktiv ist und transportieren gleichzeitig gekapselt Informationen. Dabei wird unter anderem die Information, welche ein Pubnode benötigt, um einen Ring, in welchem Publishing ermöglicht wird, aufbauen zu können, gekapselt. In diesem doppelt-verketteten Pubnodering wird Publishing durch einen store-and-forward-Ansatz der Pubnodes realisiert. Workingnodes tauschen mit Hilfe des Gossip untereinander Informationen aus, so dass die Struktur des Pubnoderings auch über die Verwaltungsbereiche der Workingnodes hinaus reicht. Kommt ein weiterer Workingnode hinzu, so wird mit einem automatischen Übergabe-Mechanismus die Verantwortung für Pubnodes mit Schlüsselpaaren in dessen Verwaltungsgebiet übergeben. Etwas umständlich ist die Einschränkung, dass alle Pubnodes nach dem Ausfall ihres Workingnodes neu beitreten müssen.

5.3.1 Pubnodes

Sobald ein Subscription-Gesuch eingereicht wird, entsteht ein passender Pubnode beim physikalischen Knoten. Ein erster sha1-Hash wird über das Schlüsselwort gebildet und ein zweiter, zufälliger sha1-Hash wird generiert. Zusammen mit einer Identifikation des physikalischen Knotens, der ip-Adresse, wird der Pubnode anhand dieses Schlüsselpaares eindeutig gemacht. Am pub/sub-System partizipieren diese Knoten erst, sobald sie von einem Verwalterknoten, auch Workingnode genannt, akzeptiert und verwaltet werden. Diese erste Beziehung zwischen Working- und Pubnode wird durch Einhaltung des Subscription-Protokolls hergestellt.

Alle Pubnodes in diesem pub/sub-System werden in einem Ring, dem Pubnodering, anhand ihres Schlüsselpaares angeordnet. Der Ring ist doppelt verkettet und ohne lange Verbindungen, die andere überspringen. Sie tragen nicht dazu bei diesen Ring aufrecht zu erhalten, sondern profitieren von Informationen, die sie regelmäßig von dem Knoten, welcher sie verwaltet, erhalten. Diese regelmäßigen Nachrichten werden zusätzlich dazu verwendet, um Ausfälle eines Workingnodes pubnodeseitig festzustellen und um zur Not ein weiteres Mal den Subscription-Prozess mit dem selben Schlüsselpaar zu durchlaufen. Den Prozess nennt man in diesem Fall dann Resubscription. Sie werden deshalb fortan als Alive-Nachrichten betituliert. Neben ihrem Verwalter kennen die Pubnodes nur zwei weitere virtuelle Knoten, ihre jeweiligen direkten Nachbarn im Pubnodering.

Diesen Pubnodes ist egal, von wem sie verwaltet werden und ob ihre Position im Ring stimmt. Sie nehmen an, sie hätten die richtigen Informationen erhalten. Einen neuen Verwalter tragen sie einfach ein, sobald ihnen von einem neuen Workingnode Alive-Nachrichten gesendet werden. Ebenso übernehmen sie, ohne die Informationen zu überprüfen, neue direkte Nachbarn. Dadurch ist es einfach möglich, bei Bedarf neue Workingnodes einzusetzen und diesen die Verantwortung für Pubnodes mit Schlüsselpaaren in ihrem Verwaltungsgebiet zu übergeben.

5.3.2 Workingnodes

Ein Workingnode ist durch ein Schlüsselpaar und eine ip-Adresse eindeutig im System identifizierbar. Anhand dieser Informationen wird er in gleich zwei Ringen angeordnet, dem Churnnodering und dem Workingnodering. Für die pub/sub-Schicht ist diese Aufteilung in zwei Ringe transparent.

Die Motivation diese Knoten einzuführen lag in der Idee, dass nicht jeder Knoten, der am System teilnimmt, auch gleichviel Arbeit zum Strukturerhalt tragen soll. In einem pub/sub-System ist das Interesse an Nachrichten und

der Subscription zu speziellen Topics wahrscheinlich nicht gleichverteilt und demzufolge sollte daraufhin die Arbeit für das System angepasst werden. So entstand die Zweiteilung von teilnehmenden Knoten in Pubnodes und Workingnodes. Hierbei stellen Pubnodes eher die Nutznießer und Workingnodes die Arbeiter dar. Workingnodes sind also die schon früher erwähnten Arbeitseinheiten bzw. Einstiegspunkte in das Gebiet eines Topic. Näheres zur fairen Lastverteilung anhand des Nutzens durch Nachrichten ist in [8] zu finden.

Durch den Beitritt in das P2P-System erhält der Knoten ein Schlüsselpaar und damit eine Position in beiden Ringen. Von diesem Zeitpunkt an kommt ein DHT-ähnlicher Ansatz zum Tragen. Ein Knoten übernimmt fortan die Verwaltung von Pubnodes, die Schlüsselpaare besitzen, welche zwischen diesem Workingnode und dem nächsten rechten Nachbarn im Workingnodering liegen. Abbildung 5.5 verdeutlicht das Prinzip. Spezialfälle dahingehend stellen der am weitesten links liegende Knoten und der am weitesten rechts liegende Knoten im Bereich eines Topic dar. Der am weitesten links Liegende übernimmt die Verwaltungsaufgaben von allen Knoten in diesem Topic mit Topicorder 0 bis zur Topicorder des nächsten rechten Nachbarn in seiner Nachbarschaftsliste der Workingnodes. Diese designtechnische Entscheidung wurde getroffen, damit jeder Workingnode für genau ein Topic Arbeit leisten muss. Der am weitesten rechts Liegende übernimmt zusätzlich das ganze Gebiet an Schlüsselpaaren zwischen sich selbst und dem nächsten rechten Workingnode, was ein relativ großes Gebiet sein kann. Der nächste rechte Workingnode hat ein Topic das um x $\in$ Hashwerte größer ist und jedes dieser x-1 dazwischen liegenden Topics hat 2^{160} Hashwerte als Topicorder. Dementsprechend hat dieser Workingnode theoretisch im bestcaste $(x-1)*2^{160}$ und im worstcase $(x+1)*2^{160}$ Hashwerte zu verwalten. Sei $\varepsilon \in \mathbb{N}$, so hat ein jeder Knoten, außer den genannten Ausnahmen, bei e $\in \mathbb{N}$ Knoten innerhalb des Gebietes, im averagecase $\frac{(1+\varepsilon)*2^{160}}{e}$ Schlüsselpaare zu verwalten.

In regelmäßigen Abständen sendet ein Workingnode allen Pubnodes, welche er zu verwalten hat, Alive-Nachrichten und wartet dann asynchron auf eine Antwort. Der Hauptgrund diese Nachrichten zu versenden liegt in der Möglichkeit Vermutungen über einen Ausfall einzelner Pubnodes anzustellen. Nach Ausbleiben einer Antwort über ein bestimmtes Zeitintervall wird ein Pubnode als ausgefallen erklärt. Die zweite Aufgabe dieser Nachrichten ist gekapselt Informationen zu transportieren. Dabei werden Informationen über die direkten Nachbarn des Pubnodes gekapselt. Lokal stellt dies für den einzelnen Workingnode kein Problem dar, da jeder Workingnode alle von ihm verwalteten Pubnodes kennt und diese deshalb sortieren kann. Global stellt dies allerdings ein Problem dar, denn ein Workingnode kann sich nicht einfach das Wissen seines linken und rechten Nachbarn verschaffen. Aus diesem

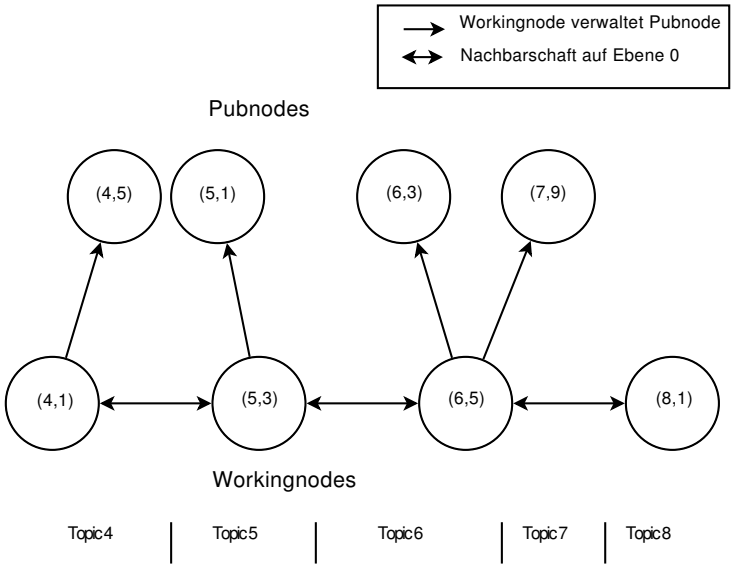

Abbildung 5.5: Verwaltungsbereiche der Workingnodes

Grund wurden die schon vorhanden Gossip-Nachrichten auf Ebene 0 erweitert. Diese kapseln nun Informationen über den nächsten Pubnode, welcher links von einem Workingnode liegt. Damit durch direkte Benachrichtigungen bei einer Änderung der Ring schneller repariert werden kann, wird der passende Verwalter zum Pubnode mitgeliefert. Erwähnenswert hierbei ist der Fakt, dass der am weitesten rechts Liegende, von einem Workingnode verwaltete Pubnode, links für alle rechten Nachfolger ist. Über den nächsten Pubnode auf der rechten Seite erhält er durch die gleiche Methodik, mit Hilfe von Gossip-Antworten, Informationen. Sinnvollerweise ist dieser Mechanismus nur in einem der Ringe, welchen der Workingnode angehört, vorzufinden.

Ein Workingnode hält eine sortierte Liste der Pubnodes, so dass er Informationen über die Sortierung an seine Pubnodes weiterleiten kann. Die Sortierung der Pubnodes ist nicht völlig trivial, da ein Workingnode kein globales Wissen besitzt. Hat er zwei oder mehr Schlüsselpaare, welche nicht im selben Topic zu finden sind, zu verwalten, so muss er entscheiden, in welcher Reihenfolge diese auf dem Ring zu finden sind. Sei Schlüsselpaar A eines Pubnodes (5,2) und Schlüsselpaar B eines anderen Pubnodes (1,8) und der Workingnode besäße das Schlüsselpaar W (5,1). Wie definiert man nun die Relation $<$ für einen Sortieralgorithmus? Die gewählte Lösung verwendet kontextsensitive Informationen über das Schlüsselpaar X des rechten Nachbarn im Workingnodering. Es gilt also $A < B$, wenn B zwischen A und X liegt. Eine ähnliche Definition anhand von W ist nicht möglich, da der am

weitesten links liegende Workingnode im Bereich eines Topics auch Knoten der Topicorder von Schlüsselpaaren zwischen 0 und seiner eigenen Topicorder verwaltet.

Wurde bei einem physikalischen Knoten die unsubscribe-Möglichkeit aufgerufen, so werden der Pubnode und all seine Workingnodes (siehe dazu [8]) entfernt. Er hört dadurch auf zu publishen und antwortet nicht mehr auf Alive-Nachrichten. Der Workingnode hat zwei Chancen mitzubekommen, wann ein Knoten gegangen ist. Zum einen wird eine direkte unsubscribe-Nachricht an den Workingnode gesendet, damit der passende Workingnode schnell den Ring aus Pubnodes reparieren kann. Falls aus einem Grund diese Nachricht nicht ankommen würde, so könnte er das Verlassen auch über den ganz normalen Timout, nach Ausbleiben von mehreren Alive-Nachrichten, feststellen. Verwaltet ab dem Zeitpunkt der Unsubscription ein Workingnode keinen Pubnode mehr sagt dieser dem nächsten rechten und linken Workingnode, welche einen Pubnode verwalten, bescheid, so dass diese relativ schnell den Pubnodering reparieren können. Der rechte Workingnode wird darüber informiert, dass der momentan linke Workingnode des nun nicht mehr als Verwalter tätigen Knotens dessen neuer linker Workingnode mit Pubnodes ist. Beim Linken ist es genau umgekehrt. Mit einem Zeitstempel versehen, werden diese Informationen nicht durch, in Gossip mitpropagierte, veraltete Informationen überschrieben. Diese Informationen bestehen bis aktuellere Informationen durch Gossip-Nachrichten propagiert werden. Um Aussagen anhand von Zeitstempeln innerhalb eines Verteilten Systems zu machen, sollten synchronisierte Uhren vorhanden sein. In einer Simulationsumgebung ist dies kein Problem, in einem realen System sollten Alternativen in Erwägung gezogen werden.

Nach welchen Regeln Workingnodes eingefügt werden ist in [8] genauer beschrieben.

5.3.3 Churnnodes

Die dritte und letzte Art der virtuellen Knoten in diesem System werden Churnnode genannt. Aus dem Namen lässt sich schon ableiten, dass sie zur Behandlung von Churn gedacht sind. Hier wird in Kürze ihr Sinn und Zweck kurz erläutert und ihre architekturelle Bedeutung erwähnt. Für genauere Informationen bezüglich der Churnnodes und des Adaptionsschritt nach dem sie eingefügt werden, ist im entsprechenden konzeptionellen Kapitel nachzulesen.

Auch Churnnodes sind, wie alle anderen virtuellen Knoten, über die erwähnte Kombination von Schlüsselpaar und ip-Adresse zu identifizieren. Ihr Platz im System ist zusammen mit den Workingnodes im sogenannten Churn-

nodering. In diesem Ring werden lookup- als auch join-Nachrichen geroutet. Dies wird in späteren Abschnitten noch genauer dargelegt. Sie sollten keinerlei Arbeit übernehmen dürfen und deshalb keine lookup-Antworten über sich selbst senden.

Die Motivation, eine andere Art von Knoten einzufügen, ist durch das gute Verhalten von GosSkip durch die Vergrößerungen von Gebieten gegeben. Je größer das Gebiet eines Topics wird, desto eher wird es schon früh durch lookup-Anfragen getroffen und desto weniger Arbeit haben die Knoten in vorhergehenden Topics bezüglich lookup- und join-Weiterleitungen.

Churnnodes sind instabil. Sie sind von Anfang an als Knoten geplant gewesen, die für eine vordefinierte Zeitspanne im GosSkip-Ring bleiben und dann wieder das System verlassen. Churnnodes haben keinerlei Aufgaben innerhalb der pub/sub-Schicht. Wie später geschildert wird, müssen einige Vorkehrungen getroffen werden, da Churnnodes keinerlei Verwaltungsaufgaben übernehmen. Theoretisch könnte man sie in einer späteren Arbeit dazu verwenden, eine gewisse Ausfallsicherheit zu realisieren, indem sie nach dem Ausfall von Workingnodes erstmal deren Arbeit übernehmen. Bei solchen Überlegungen muss man allerdings beachten, dass Churnnodes instabil sind und deshalb dabei auch in Workingnodes gewandelt werden müssten.

5.3.4 Subscription

Der Subscribtion Prozess beginnt damit, dass lokal, bei physikalischen Knoten ein Pubnode erzeugt und anschließend zu einer Liste aus Pubnodes hinzugefügt wird. Anhand des ihm zugewiesenen Schlüsselpaares beginnt eine Suche. Um nach Erhalt der Antwort auf die Suche noch zu unterscheiden, für was der lookup ausgeführt wurde, kann man anhand einer Number used Once (NONCE) verschiedene parallele lookups für einen virtuellen Knoten unterscheiden. Dies ist durch alternative Konzepte, wie eingefrohrene Bezeichner, noch verfeinerbar. Anhand eines softstate-Ansatzes wird die lookup-Nachricht, sobald ein Timeout abgelaufen ist, als verloren gehandhabt und mit der selben NONCE noch einmal versendet.

Man sendet nun eine Subscription zu dem als Antwort auf die Suche zurückgelieferten Workingnode. Der Timeout für die lookup-Anfrage sollte dann gestoppt werden, damit keine konkurrierenden Aktionen durchgeführt werden. Es ist nun elementar wichtig den Timout für Alive-Nachrichten zu starten. Erhält nämlich der Pubnode in diesem Zeitintervall keine Alive-Nachricht, ist der gefundene Workingnode wahrscheinlich ausgefallen und der Prozess muss mit der Suche erneut begonnen werden.

Erhält ein Workingnode das Subscription-Gesuch, so fügt er den anfragenden Pubnode, als von ihm verwaltet in eine Liste mit allen Pubnodes, für

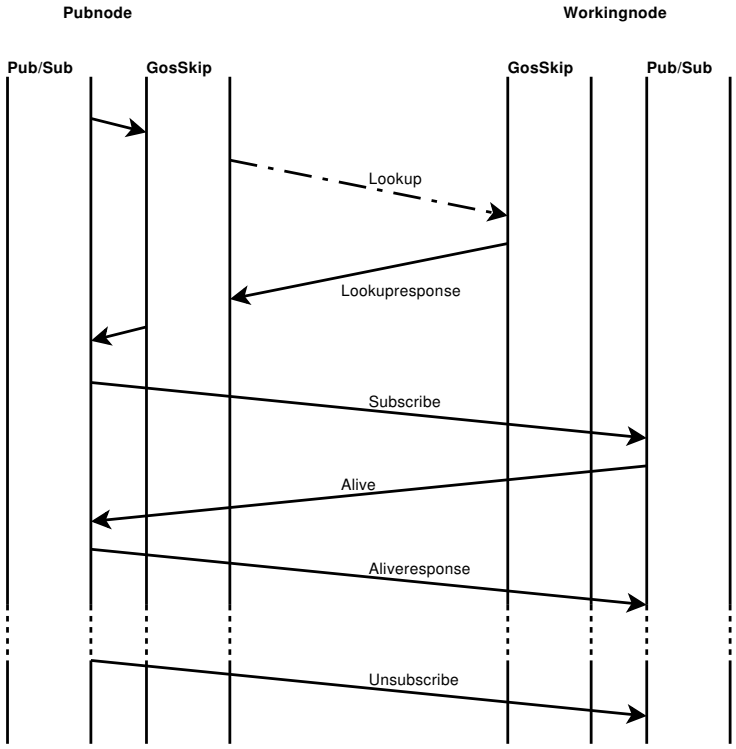

Abbildung 5.6: Subscription Protokoll

die er arbeitet, ein und sendet sofort eine außerplanmäßige Alive-Nachricht an diesen. Hat ein Workingnode auch die Nachbarn des Pubnodes zu verwalten, schickt er diesen ebenfalls außerplanmäßige Alive-Nachrichten, um sie über den neuen Pubnode zu informieren. Ist der neue Pubnode am rechten und/oder linken Rand der vom Workingnode verwalteten Pubnodes, wird durch eine direkte Benachrichtigung der Nachbarn eine zeitnahe Integration in den Pubnodering ermöglicht. Ab diesem Zeitpunkt gilt ein Pubnode als ins System integriert.

5.3.5 Publish

Hier wird ein store-and-forward-Ansatz zum Publishen verwendet und ist dazu gedacht, in einem Gebiet Informationen zu verbreiten. Dadurch, dass die Pubnodes sich wegen ihres Schlüsselpaares, das sich bei gleichem Interesse im ersten Teil des 2-Tupels gleicht, in der selben Nachbarschaft befinden, werden nur Knoten, die auch wirklich Interesse an den Ereignissen haben, zum Speichern und Weiterleiten der Nachrichten gezwungen. Ein getaktetes Fluten innerhalb des Ringes bis an den Rand des Bereiches eines Topic, wurde

zur Nachrichtenauslieferung verwendet.

Eine Vereinfachung wurde getroffen, indem ein Publisher dem Topic, in welchem er Nachrichten verbreiten möchte, beitreten muss. Ein angedachter Ansatz hätte die lookup-Nachricht verwendet, um einen beliebigen Knoten als Einstiegspunkt zu finden und von dort aus die Publish-Nachricht weiterzuleiten. Lookup wird allerdings als Arbeit für die Subscription angesehen und ist deshalb für das Fairnesskriterium zu beachten. Demzufolge würde diese Art des Publishens das Ergebnis zur Evaluierung von Fairness in Bezug auf Churn verfälschen. Andere Ansätze könnten darauf beruhen, in seiner Nachbarschaftstliste nach einem Einstiegspunkt für das Topic zu suchen. Diesem Ansatz könnte noch mehr Beachtung geschenkt werden. Dies war aber für diese Arbeit und [8] irrelevant und wurde deshalb nicht weiter verfolgt.

Wie in [8] genauer beschrieben, sind gewisse fehlertolerante Ansätze bezüglich der Nachrichtenauslieferung vorgesehen worden.

5.3.6 Routing im Churnnodering

Durch Einfügen von Churnnodes in das Gebiet eines Topics wird dieses größer. Mit hoher Wahrscheinlichkeit wird also auch ein Churnnode, bei der Suche nach einer bestimmten Identifikation für die Subscription zurückgeliefert. Da dieser keine Verwaltungsaufgaben übernimmt, ist es unpraktikabel, wenn ein Churnnode Informationen über sich zurückliefert. Stattdessen ist es sinnvoller, wenn er Informationen über den nächstbesten Workingnode in seiner Nähe zurückliefern würde. Diese Informationen besitzt er per se nicht.

Gossip-Nachrichten fließen in diesem System immer. Der Grundgedanke war nun, Informationen über den nächsten Workingnode in der Nähe eines Churnnodes mitpropagiern zu lassen. Nach rechts geschieht dies mit Hilfe der Gossip-Nachrichten auf Ebene 0. Die jeweiligen Workingnodes fungieren als Stopper. Sobald die Informationen über einen linken Workingnode, bei einem weiter rechts liegenden Workingnode angelangt sind, werden diese nicht mehr weiterpropagiert. Churnnodes können nun Informationen über den nächsten linken Workingnode zurückliefern, welcher ja als Verwalter für Schlüsselpaare zwischen dem eigenen Schlüsselpaar und dem des nächsten rechten Workingnodes eingesetzt ist.

Die direkte Zuordnung zu einem Topic ist so aber noch nicht gewährleistet. An den Grenzen der einzelnen Gebiete für ein Topic werden trotzallem fehlerhafte Informationen zurückgeliefert. Man stelle sich beispielsweise folgendes Szenario vor. Es wird nach dem Schlüsselpaar (5,7) gesucht. Ein Churnnode C besäße das Schlüsselpaar (5,1), sein linker Workingnode-Nachbar W1 (3,8) und sein rechter Workingnode-Nachbar W2 (5,9). Nach der obigen Vorschrift hätte C nur über W1 mit (3,8) Informationen und würde diese zurückliefern.

Da man aber in den Bereich des Topics mit Hashwert fünf hineintreffen wollte, wären Informationen über den Knoten W2 mit (5,9) richtig gewesen.

Um den gerade eben dargelegten Fehler zu vermeiden, wurden nun Informationen über den rechten Workingnode mit den Response-Nachrichten von Gossip mitgeliefert. Auch hier fungieren weiter links liegende Workingnodes als Stopper. Somit hält ein jeder Churnnode neben Informationen über den linken Workingnode auch Informationen über den rechten Workingnode.

5.3.7 Handover

Durch Churn im Workingnodering werden die Bereiche, in denen Schlüsselpaare verwaltet werden müssen, immer neu aufgeteilt. Der Handover-Prozess wurde integriert, damit die Übergabe der Verwaltung von Pubnodes zwischen Workingnodes in ein, vom entsprechenden Workingnode betreutes Gebiet, automatisiert stattfinden kann.

Bei jeder Gossip-Nachricht und bei jeder Gossip-Antwort auf Ebene 0 wird automatisch ein Entscheidungsalgorithmus, welcher überprüft, ob nach rechts oder links Knoten übergeben werden müssen, angestoßen. Dies geschieht direkt nach diesen zwei Nachrichten, da hierbei neuere Informationen über die nächsten Workingnodes vorbeigeflossen sind. Entscheidungen werden auch hier wieder anhand der Schlüsselpaare getroffen.

Nach folgenden Regeln wird die Übergabe durchgeführt. Der momentane Verwalter erhält Gossip-Nachrichten von einem Nachbarn auf Ebene 0 und überprüft nun $\forall$ verwaltete Pubnodes:

1. Hat ein Pubnode das gleiche Topic wie der Nachbar, aber der momentane Verwalter nicht, dann erfolgt die Übergabe.

2. Haben weder der Pubnode, der momentane Verwalter noch der Nachbar das gleiche Topic, schaue, ob das Topic des Pubnodes nicht zwischen dem des Verwalters und des Nachbarn liegt. Falls dies der Fall ist erfolgt die Übergabe.

3. Haben zwar der momentane Verwalter und der Nachbar das gleiche Topic, aber der Pubnode nicht, so entscheide äquivalent zu 2.

4. Haben sowohl Pubnode, momentaner Verwalter und Nachbar das gleiche Topic, dann ermittle zuerst, ob die Topicorder des Verwalters oder die Topicorder des Nachbarn größer ist.

   (a) Ist die Topicorder des momentanen Verwalters größer, übergebe, wenn die Topicorder des Pubnodes kleiner als die des momentanen Verwalters ist.

(b) Ist die Topicorder des Nachbarn größer, übergebe, wenn die Topicorder des Pubnodes größer als die des Nachbarn ist.

5. In allen anderen Fällen behält der momentane Verwalter den Pubnode.

Im ersten Moment mag die Regel vier schwierig wirken. Sie ist aber leicht zu verstehen, wenn man einen Zahlenstrahl von 0 bis max $\in \mathbb{N}$ heranzieht. Zwei beliebige Zahlen i,j $\in \mathbb{N}$ seien darauf angereiht, wobei $i > j$. Eine dritte Zahl k $\in \mathbb{N}$ sei beliebig auf dem Zahlenstrahl zu verteilen. Sie ist j zuzuweisen, wenn i lokal entscheidet $k < j$. Das wäre Fall 4(a). Sie ist i zuzuweisen, wenn j lokal entscheidet $k > i$. Das wäre Fall 4(b).

5.3.8 Workingnodering

Wie in vorherigen Absätzen erwähnt ist der Workingnodering ein GosSkip-Ring, in dem Workingnodes angereiht sind. Der einzige Zweck diesen aufzubauen und zu halten lag in der Möglichkeit, gleichverteilt über die Menge der Knoten, zufällige Knoten suchen zu können. Außerdem bietet er die Möglichkeit, Aussagen über die geschätzte Anzahl der Teilnehmer machen zu können. Angestrebt werden sollte eine x:1 Abbildung von Pubnodes auf Workingnodes, wodurch die Anzahl der Pubnodes im Vergleich zu den Workingnodes mit $x* \mid Workingnodes \mid$ zu beaufschlagen wäre.

Der Churnnodering aus Working- und Churnnodes existiert bereits und wird für join- und lookup-Nachrichten verwendet. Die Idee war nun, keine zweite join-Nachricht im Workingnodering zu propagieren, um mögliche Inkonsistenzen, die durch erfolgreichen join in den einen und gleichzeitigen Verlust der join-Nachricht in dem anderen Ring, entstehen würden, zu vermeiden. Die Lösung war den join-Prozess nur im Churnnodering zu durchlaufen. Der Workingnodering könnte nun von den sowieso schon fließenden, zusätzlichen Informationen (siehe Kapitel 5.3.6) profitieren.

Erhält ein Workingnode im Churnnodering Informationen durch Gossip-Nachrichten über einen Workingnode, der links von ihm liegt, kann dieser die Information dazu verwenden, ihn als linken Nachbarn auf Ebene 0 im Workingnodering einzutragen. Durch Gossip-Antworten verläuft der Prozess gerade umgekehrt, denn da bekommt der Workingnode Informationen über seinen rechten Nachbarn im Workingnodering.

5.3.9 Anfragen

Um Informationen von anderen Knoten zu erhalten und dennoch nicht alle Knoten im System anfragen zu müssen, da dies nicht mit der Anzahl der

Knoten skalieren würde, wurde eine Möglichkeit bereit gestellt, stichprobenhaft Anfragen zu stellen. Durch diese Anfragen erhält ein Knoten Informationen über andere Teilnehmer des Systems, welche beispielsweise die Höhe des Churns bei diesen Knoten im letzten Intervall und deren Einschätzung über die Anzahl der Knoten in ihrem Gebiet sein könnten.

Als erstes muss hier die Größe des Systems abgeschätzt werden. Wie schon erwähnt, erhält man eine Einschätzung der Systemgröße anhand des höchsten Eintrages in der Nachbarschaftsliste. Eine zufällige und vernünftige Anzahl dieser Knoten soll nun angefragt werden. Ein Ansatz wäre beispielsweise lograithmisch viele anzufragen. Dazu muss die Möglichkeit des GosSkip-Ringes, einen Nachbarn, der genau x hops weit auf dem Ring der Ebene 0 entfernt ist, zu finden, verwendet werden. Mit einer NONCE versehen können diese lookup-Anfragen von anderen lookup-Anfragen unterschieden werden. Durch die Antworten kann man die Knoten, welche als Antwort zurückgeliefert wurden, direkt nach den Stichproben, welche man erhalten möchte, anfragen.

Durch die Abschätzung der Größe des Systems erhält man einen Fehler $\varepsilon \in \{f \mid f \in \mathbb{N} \wedge 0 \leq f < k^{max+1} - k^{max}\}$, wobei max die Höhe der Nachbarschaftslisten und k die Skiplänge ist. Das Problem hierbei ist, dass der Ring noch Knoten zwischen der höchsten langen Verbindung und dem Knoten, der angefragt hat, besitzt. Würde man stattdessen einfach k^{max+1} potentielle Knoten anfragen, so fragt man zu viele Knoten an. Deshalb wurde die Möglichkeit gewählt, mit der Wahrscheinlichkeit 0.5 nach links oder rechts Anfragen zu stellen. Durch diese Möglichkeit werden alle Knoten im System als mögliche Ziele ausgewählt.

Dennoch werden die Anfragen ungleichmäßig verteilt. Knoten in der Nähe des Anfragenden werden nicht so häufig getroffen, als Knoten, die weiter entfernt liegen. Alle Knoten, die in dem oben genannten Fehlerbereich liegen, kommen nur bei einer Wahrscheinlichkeit von 0.5 als Ziel in Frage. Alle Knoten außerhalb des Fehlerbereiches kommen mit einer Wahrscheinlichkeit von 1 als Ziel in Frage. Der Fehler ist nicht schlimm, da Knoten in der engeren Nachbarschaft meist im gleichen Gebiet liegen und deshalb nicht so häufig angefragt werden müssen. Informationen aus dem eigenen Bereich eines Topics verbessern in geringem Umfang die lokale Sicht, bieten aber keinen bessern Ausblick für die angenäherte globale Sicht.

Kapitel 6

Evaluation

6.1 Netzwerksimulator

Netzwerksimulatoren bieten die Möglichkeit, mit wenig Aufwand große Verteilte Systeme zu testen. Sie ersparen einem ein reales Netzwerk aus ein paar tausend Rechnern aufzubauen und nach jeder Änderung diese auf all jenen aufzuspielen. Stattdessen muss man nur den Simulator parametrieren. Die Frage danach, mit welchem Simulator man ein bestimmtes System testen will, ist nicht einfach beantwortet. Es gibt viele Faktoren, die es zu beachten gilt und die, je nachdem, mit welcher Intention man diese betrachten möchte, auch stark variieren. [25] haben sich dazu Gedanken gemacht und verschiedenste P2P-Simulatoren unter Gesichtspunkten wie z.B.: bietet die Simulator Architektur Möglichkeiten realistische Verzögerungen in die Simulation mit einzubeziehen, kann man damit die Skalierbarkeit eines Systems testen bzw. gibt es genug Dokumentation, um den Simulator zu betreiben, untersucht. Die meisten Simulatoren hatten jedoch sehr viele Mängel, wie mangelnde Dokumentation und mangelnder Support, sowie fehlende Möglichkeiten statistische Auswertungen zu erfassen. Das Oversim-Overlay-Simulation-Framework [3] des Institute of Telematics der Universität Karlsruhe hat die Zielsetzung all jene Mängel zu bewältigen und ist damit als Simulations-Tool gewählt worden. Oversim ist ein Framework für das Objektorientierte, C++ basierte OMNet++ Simulationssystem [26].

6.1.1 Omnet++

Omnet++ bietet die Möglichkeit, parallele und verteilte Systeme, insbesondere Netzwerke, ereignisbasiert zu simulieren. Ereignisbasiert bedeutet, dass eine Liste mit Ereignissen existiert, welche nach und nach abgearbeitet wird, die sogenannte Future Event List (FEL) . Alle Ereignisse werden anhand des

simulierten Zeitstempels einsortiert und ausgeführt. Es wird davon ausgegangen, dass zwischen den Ereignissen nichts interessantes passiert.

Um mit Hilfe von Omnet++ Simulationen zu starten, müssen Module definiert werden, in denen auf Ereignisse reagiert bzw. in denen Ereignisse kreiert werden. Diese Verhalten wird in C++ Klassen, welche von der Klasse cSimpleModule abgeleitet sein müssen, festgelegt. Module können geschachtelt sein und so eine unendlich große hierarchische Struktur bilden.

NED ist eine Sprache, mit deren Hilfe man die Topologie des simulierten Systems, sowie einfache, als auch geschachtelte Module beschreiben kann. Diese Definitionen werden in ".ned"-Dateien geschrieben und können dynamisch in das Simulationsprogramm geladen werden. Alternativ kann daraus, durch das sogenannte nedtool, einem Perl Skript, eine C++ Klasse generiert werden. Die einzelnen Module kommunizieren über Nachrichten.

Nachrichten können dazu verwendet werden um einzelne Pakete, Rahmen oder ähnliches in einem Netzwerk zu simulieren. Ebenso ist durch verzögerte Nachrichten der Module an sich selbst die Möglichkeit gegeben Taktungen und zeitbasierte Ereignisse zu implementieren. Einzelne Nachrichten können von der Klasse cMessage abgeleitet und dann in C++ implementiert werden. Alternativ kann man in .msg-Dateien mit Hilfe der *MSG*-Sprache die Nachrichten deklarieren und mit Hilfe eines Perl-Skripts (opp_msgc) C++ Klassen automatisch generieren lassen.

Ein graphisches Frontend ist in TKENV implementiert, welches in Abbildung 6.1 exemplarisch dargestellt wurde. Dieses kann dazu verwendet werden um das Debugging zu erleichtern. Durch diese Grafische Benutzerschnittstelle (GUI) ist es möglich Nachrichten zu verfolgen, die Topologie des Systems zu verifizieren oder gar Parameter während der Simulation zu beobachten und zu verändern.[26, 27]

6.1.2 OverSim

OverSim [3] ist ein Framework, welches Simulationen von P2P-basierten Systemen mit Omnet++ [26] vereinfacht. Ein schichtenartiger Aufbau, in Abbildung 6.2 dargestellt, bietet die Möglichkeit nur einzelne Elemente transparent auszutauschen.

Die Underlay Schicht bietet in der aktuellen Version drei Möglichkeiten ein Netzwerk mit DCE's (Data Circuit-Terminating Equipment) undf DTE's (Data Terminal Equipment, Datenendeinrichtung) aufzubauen. Zum einen gibt es das INET Underlay, welches einem die Möglichkeit gibt, einen kompletten simulierten Netzwerk-Stack zu nutzen. Des Weitern gibt es das Single-Host Underlay, mit dem man, wie in PlanetLab, einzelne Kommunikationsendpunkte über ein reales Netzwerk verbinden kann. Als letzte Möglichkeit

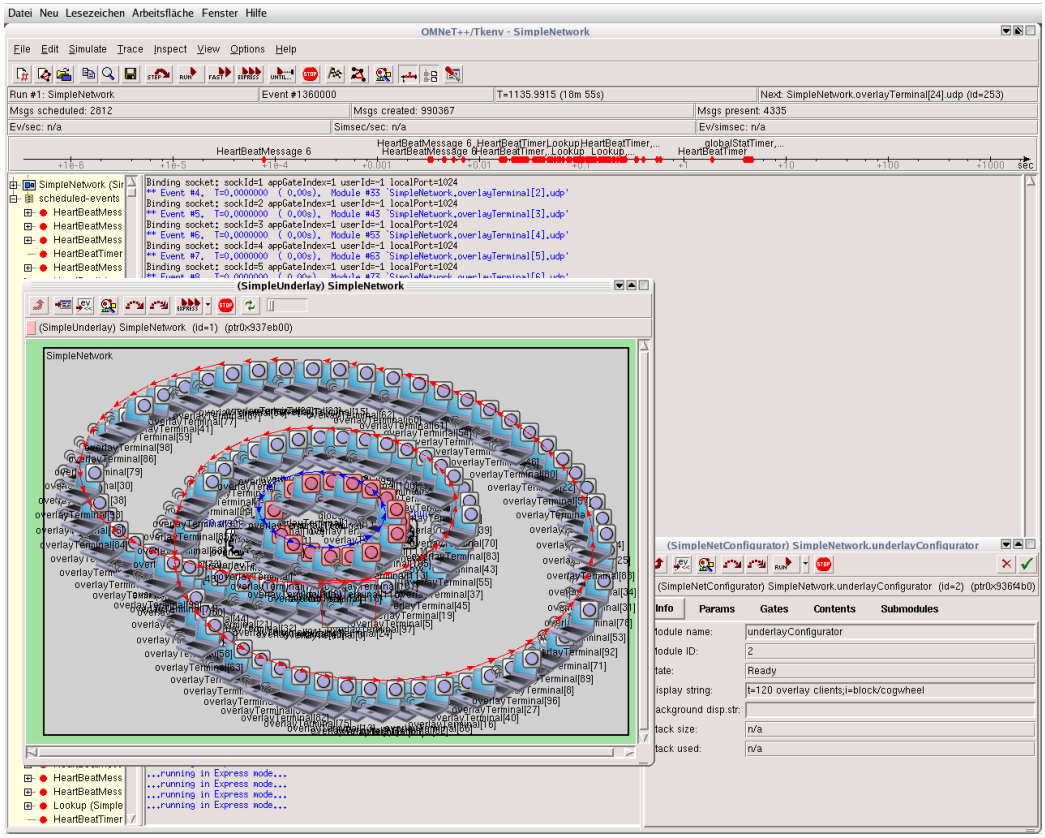

Abbildung 6.1: Screenshot einer Simulation des dynamischen pub/sub Systems

gibt es die schlanke Variante des Simple Underlays, in der Kommunikationsendpunkte über eine globale Routing Tabelle kommunizieren können. Das Simple Underlay wurde für diese Arbeit gewählt, da für ein reales Netzwerk mit dem Single-Host Underlay zuviel Rechner und für eine realistischere Simualtion mit dem INET Underlay zuviel Events gibt und damit zuviel Rechenzeit verbraucht würden.

Die Overlay Schicht ist dazu gedacht, um verschiedenste P2P-Protokolle zu implementieren. Ein Omnet++ spezifisches und für Overlays geeignetes graphisches Frontend ermöglicht es die Struktur des P2P-Systems zu visualisieren. Wie in Abbildung 6.1 dargestellt, kann man mit Hife von Pfeilen die Overlay-Struktur eines solchen Netzes visualisieren. Chord [15], Pastry [21] und einige andere Protokolle sind für OverSim schon implementiert. Um ein eigenes P2P-Protokoll zu implementieren sind im Grunde drei Dateien notwendig. Zum einen eine ned-Datei mit den entsprechenden Angaben für das Omnet++ spezifische Modul. Zum anderen eine optionale msg-Datei um

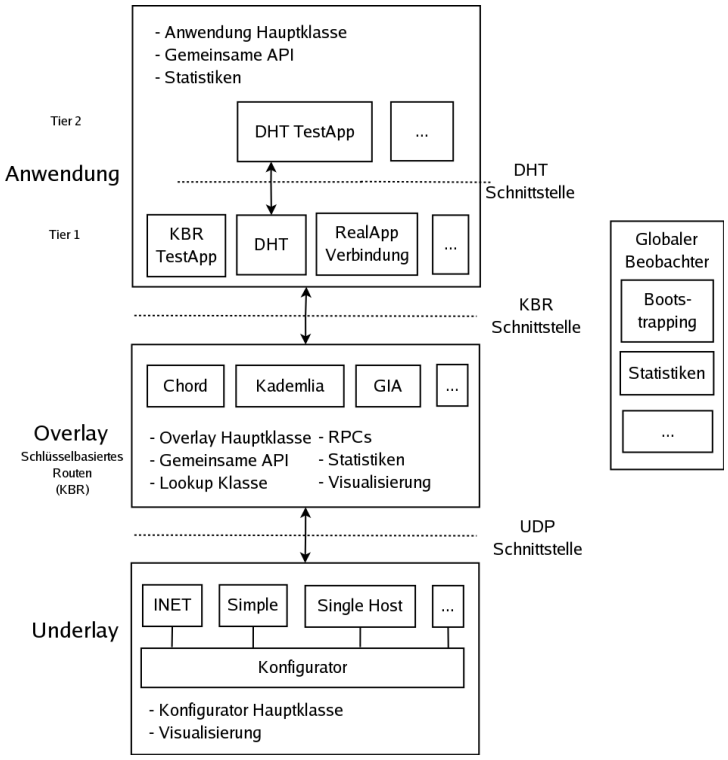

Abbildung 6.2: Modulare Struktur von Oversim [3]

Nachrichten, die zwischen den Peers ausgetauscht werden, zu deklarieren. Und schlussendlich eine C++ Klasse, in welcher die eigentliche Implementation, die Reaktion auf Ereignisse, vorgenommen wird. Diese Klasse wird von der Klasse BaseOverlay abgeleitet, welche OverSim zur Verfügung stellt. Aus der Basisklasse zieht ein Peer Vorteile dahingehend, dass diese schon Funktionen zur Verfügung stellt, um transparent Nachrichten via UDP zu versenden oder automatisch eine ip-Adresse zugewiesen wird. Für diese Arbeit wurde auf dieser Schicht GosSkip implementiert.

Die Application Schicht besteht aus verschiedenen Tiers. Aktuell ist es möglich bis zu drei Tiers auf das Overlay aufzusetzen. Um mit der Overlay Schicht zu kommunizieren gibt es einen Pointer auf das Overlay, eine Common-API Schnittstelle oder eine Nachrichten-basierte Schnittstelle. Ebenso wie auf der Overlay Schicht, benötigt man hier eine ned-Datei, eine optionale msg-Datei, sowie eine C++ Implementierung. Allerdings sind diese für jedes benötigte Tier anzulegen. Um Kommunikation zwischen den Application-Ebenen der Peers zu betreiben musste eine eigene Möglichkeit implementiert werden gekapselte Nachrichten über die Overlay Schicht umzuleiten, über das Netzwerk zu transportieren und über die Overlay Schicht

des Zielsystems auf die Application-Ebene zu schicken. Diese Möglichkeit, Inter-Tier-Kommunikation über das simulierte Netzwerk zu betreiben, wurde in einer aktuelleren Version von OverSim auf ähnliche Weise, wie in der für diese Arbeit implementierten Version, nachträglich hinzugefügt. Das pub/sub-System wurde auf der Tier 1 Schicht implementiert.

Bootstrapping bezeichnet die Technik, einen Knoten, der schon im Netzwerk ist, zu finden. Um Bootstrapping bei OverSim zu betreiben existiert ein BootstrapOracle, welches einen zufälligen Knoten im Netzwerk zurückliefern kann. Für GosSkip wurde das BootstrapOracle erweitert, so dass es virtuelle Knoten und nicht physikalische Knoten zurückliefert. Damit wurde gewährleistet, dass die Wahrscheinlichkeit, die verschiedenen virtuellen Knoten im GosSkip-Ring zu treffen, gleichverteilt ist.

Für diese Untersuchungen bezüglich Churn ist es wichtig, dass man vom BootstrapOracle gleichverteilt über das bestehende Netzwerk Knoten zurückgeliefert bekommt. Wäre dies nicht der Fall, so hätte man einen oder mehrere Knoten, die durch Subscriptions übermäßig belastet würden, denn für jede Subscription besorgt sich ein jeder Knoten vom BootstrapOracle einen zufälligen Einstiegspunkt in das System.

6.2 GosSkip

Um das Verhalten von GosSkip bezüglich des Churns in bestimmten Gebieten des Ringes, also dort, wo Knoten, die dem selben Topic angehören gruppiert sind, zu untersuchen, wurde ein simpler GosSkip-Ring herangezogen. Die Anordnung der Knoten geschah nachdem in dieser Arbeit vorgestellten Prinzip des Schlüsselpaares. In dem Ring wurden Subscriber anhand des Schlüsselpaares, zuerst nach dem Topic und dann nach der Topicorder, angeordnet. Während der Simulation ist jede Sekunde ein Knoten entstanden und ist zu einem von zehn möglichen Topics beigetreten. Nachdem eine Grundlast von 200 Knoten erreicht war, wurde Churn durch Subscriptions und Unsubscriptions im Topic mit dem Hashwert fünf verursacht. Einjeder Knoten entschied alle fünf Sekunden mit einer Wahrscheinlichkeit von 0.1, ob er das Topic fünf verlassen möchte. Nach der Unsubscription wurde jeder Knoten in einen schlafenden Zustand versetzt, in welchem er keine Nachrichten mehr versenden konnte. Einjeder Knoten, der sich in einem solchen Zustand befand, entschied alle fünf Sekunden mit einer Wahrscheinlichkeit von 0.5, ob er dem Topic fünf beitritt. Durch dieses Verhalten der Knoten entstand Churn. Das Ergebnis der Simulation ist in 6.3 dargestellt. Der Graph stellt die Arbeit die jedes Topic durch Churn leisten muss, dar. Wie man sieht, wurden sehr viele Topics vor dem belasteten Topic fünf durch zusätzliche Arbeit schwer

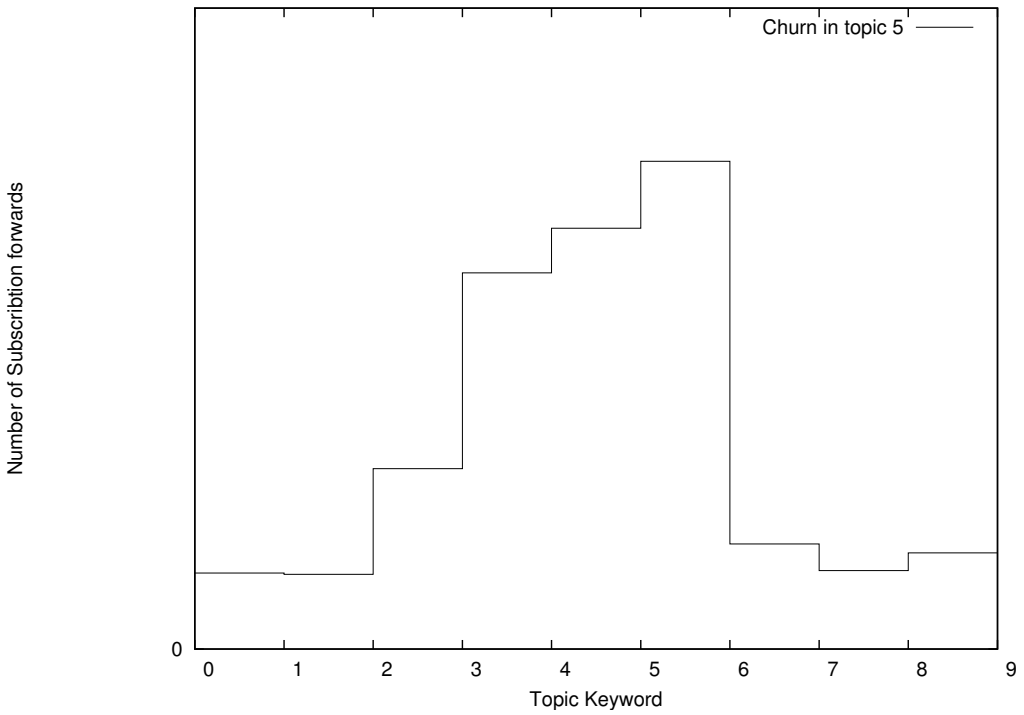

Abbildung 6.3: Verhalten von GosSkip bei Subscriptions und Unsubscriptions

beansprucht. Die Erklärung für dieses schlechte Ergebnis liegt darin, dass die Knoten keine Zeit hatten lange Verbindungen nach Topic fünf aufzubauen, bzw. häufig alte und kaputte lange Verbindungen verwendet wurden.

Ein besseres Verhalten zeigte sich, als Churn nur durch Subscriptions verursacht wurde. Ein ebenfalls aus 200 Knoten bestehender und zufällig über zehn Topics verteilter Ring wurde aufgebaut. Nachdem dieses System stand wurde sekündlich ein Knoten generiert. Dieser Knoten trat Topic fünf bei. In Abbildung 6.4 ist das Ergebnis dargestellt. Dadurch, dass das Gebiet in Ruhe wachsen konnte, wurden auch sehr viele lange Verbindungen generiert. Diese waren durch den hohen Churn nicht immer korrekt mit dem 2^i-ten Nachbarn verbunden, aber sie ermöglichten ein rasches Eintreffen in Topic fünf. Aus der Abbildung ist ebenfalls abzulesen, dass eine gewisse Grundlast in Topic vier immer mitgetragen werden muss. Egal wie groß Topic fünf ist, das Topic vier enthält die meisten und kürzesten Verbindungen nach Topic fünf. Ebenso kommt hinzu, dass der letzte Knoten in Topic vier für den Schlüssel-Bereich zwischen seinem Schlüsselpaar und dem des ersten Knoten in Topic fünf Knoten eingliedern musste.

Um das Verhalten von GosSkip bezüglich der Größe von Bereichen zu

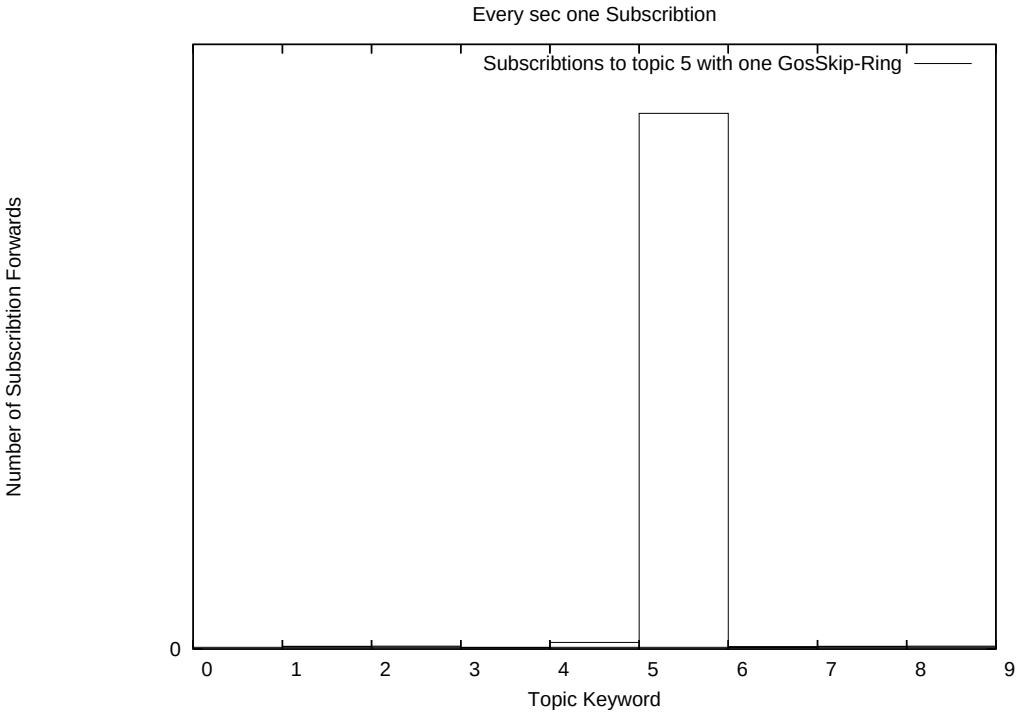

Abbildung 6.4: Verhalten von GosSkip bei Subscriptions

evaluieren, wurden zwei Versuche durchgeführt. Sie sollen zeigen, dass, egal wie groß ein Bereich ist, die Gleichverteilung der Verbindungen bestehen bleibt und dafür sorgt, dass diese Eigenschaft für eine Gleichverteilung der Arbeit durch lookup- und join-Anfragen sorgt. Churn wurde durch lookup-Nachrichten simuliert, da diese für die Subscriptions des vorgestellten Systems verwendet werden.

Im ersten Versuchsaufbau wurden 200 Knoten exakt gleichverteilt in zehn Topics gestreut. Diese zehn Topics werden als Topic null bis neun benannt. Absolut gleichverteilt waren auch die Anfragen. Alle fünf Sekunden wurde ein lookup von jedem eingefügten Knoten, in dessen Topic und mit dessen Topicorder, versendet. Als Startpunkt der Suche diente jeweils ein zufälliger Knoten aus dem BootstrapOracle. Der Versuch lief über einen simulierten Zeitraum von 24 Stunden. Die Verteilung der Knoten auf die Topics ist in Spalte eins und zwei von Tabelle 6.1 zu sehen. Die Arbeit lookup-Nachrichten zu erhalten und zu verarbeiten, wurde während der Simulation gemessen. Das Ergebnis dieser Messung ist in der dritten Spalte eingetragen. Wie zu erwarten mussten alle Topics gleichviel Arbeit leisten um lookup Nachrichten weiterzuleiten. Den Durchschnittswert an Arbeit pro Knoten, welcher in der vierten Spalte zu finden ist, erreichte jeder Knoten im System bis auf

eine geringe Abweichung. Dies bedeutet, dass selbst jeder einzelne Knoten innerhalb der Topics in etwa gleichviel Arbeit leisten musste. In Bezug auf Churn ist das Ganze fair, denn jedes Topic musste proportional zum Churn gleichviel Arbeit leisten. Das Verhalten bezüglich Churn wurde mit der vorgestellten Metrik ausgewertet. Die Spalte fünf zeigt den Fairnessquotienten. Das Verhalten des Systems ist sehr gut, da diese Werte um maximal 0.01 differieren.

Topic	Knotenanzahl	lookup	lookup pro Knoten	Fairness
0	20	1607663	80383.15	0.21
1	20	1608317	80415.85	0.21
2	20	1606379	80318.95	0.22
3	20	1606308	80315.4	0.22
4	20	1605891	80294.55	0.22
5	20	1608728	80436.4	0.21
6	20	1606345	80317.25	0.22
7	20	1607515	80375.75	0.21
8	20	1607622	80381.1	0.21
9	20	1607399	80369.95	0.22

Tabelle 6.1: Ergebnis bei einer gleichmäßigen Verteilung von Knoten

Der nächste Versuch zeigt, wie sich das System bei ungleichmäßig befüllten Topics verhält. Es ergab sich eine zufällige Verteilung der Knoten, wie sie in Tabelle 6.2 zu sehen ist. Die Gesamtzahl der Knoten betrug hier ebenso 200. Ein jeder Knoten stellte eine lookup-Anfrage nach seinem Topic und seiner Topicorder. Der Start dieser lookup-Anfragen wurde hier ebenfalls zufällig durch das BootstrapOracle bestimmt. Die ganze Simulation lief einen simulierten Tag. Während der Simulation hat ein jeder Knoten gezählt, wieviele lookup-Nachrichten er erhalten hat. Dabei sind sowohl diejenigen Nachrichten berücksichtigt worden, die für ihn wichtig waren, als auch diejenigen, welche er weiterleiten musste. Das Ergebnis ist in der dritten und vierte Spalte von Tabelle 6.2 zu sehen. Bereiche von Topics, die größer waren, wurden häufiger getroffen als kleinere. Die durchschnittliche Anzahl an zu verarbeitenden lookup-Nachrichten pro Knoten liegt allerdings immer in etwa in der selben Größenordnung. Tatsächlich schwankte die exakte Trefferzahl pro Knoten in einem kleinen Epsilon-Bereich um den Durchschnittswert. Der Fairnessaspekt ist in der letzten Spalte der Tabelle evaluiert worden. Anhand der definierten Metrik kann man Aussagen, dass das Verfahren fair ist. Auch hier unterscheidet sich der Fairnessquotient um maximal 0.1. Da, je nach Größe der Topics, unterschiedlich häufig lookup-Anfragen dorthin gestellt wurden, ist hier genau die Eigenschaft gezeigt, welche das Einfügen von

Knoten, also Churnnodes oder Workingnodes, als Problemlösung untermauert.

Topic	Knotenanzahl	lookup	lookup pro Knoten	Fairness
0	19	1527552	80397.47	0.21
1	23	1848493	80369.26	0.22
2	12	964876	80406.33	0.21
3	23	1848606	80374.17	0.21
4	24	1928224	80342.67	0.22
5	24	1928384	80349.33	0.22
6	20	1606975	80348.75	0.22
7	21	1687817	80372.24	0.22
8	24	1929491	80395.46	0.21
9	10	802208	80220.8	0.22

Tabelle 6.2: Ergebnis bei einer ungleichmäßigen Verteilung von Knoten

6.3 Grundlast

In jedem pub/sub-System gibt es eine gewisse Grundlast an Arbeit, um die Struktur aufzubauen und sie zu erhalten. Ein jeder Knoten muss diese Grundlast handhaben. Sie ist im P2P-System in den GosSkip-Ringen zu finden, denn es ist die Last, die durch Gossip-Nachrichten und Antworten gegeben ist. Join-Nachrichten weiterzuleiten ist Arbeit, die vom Churn im GosSkip-Ring abhängig ist. Die Last, die speziell durch Subscriptions verursacht ist, entsteht durch lookup-Nachrichten. Auf pub/sub-Ebene fallen vorallem die regelmäßigen Alive-Nachrichten und Alive-Antwort-Nachrichten ins Gewicht. Alle anderen Nachrichten sind vernachlässigbar, da sie nicht annähernd so häufig auftreten.

In dem verwendeten System wurden alle fünf Sekunden Gossip-Nachrichten auf allen Ebenen gesendet und auf jede dieser Nachrichten geantwortet. Die Grundlast des Systems ist dadurch hoch. Durch diese Häufigkeit wird allerdings eine gewisse Sicherheit bezüglich hohem Churn innerhalb des GosSkip-P2P-Ringes gewährleistet. Der Mechanismus, der Ausfälle von Nachbarn erkennen lässt, hängt nämlich direkt von Gossip-Nachrichten, bzw. dem Ausbleiben von Antworten ab. Ist die Rate an Ausfällen zu hoch, bzw. die Rate der Nachrichten, anhand derer Ausfälle festgemacht werden, zu gering, so könnte dies zu einem Gesamtausfall des Systems führen. Nach einer bestimmten Zeit, in der keine Antworten eintreffen, wird ein Nachbar als ausgefallen vermutet. Eine adaptive, an Churn im GosSkip-Ring anpassbare, Lösung

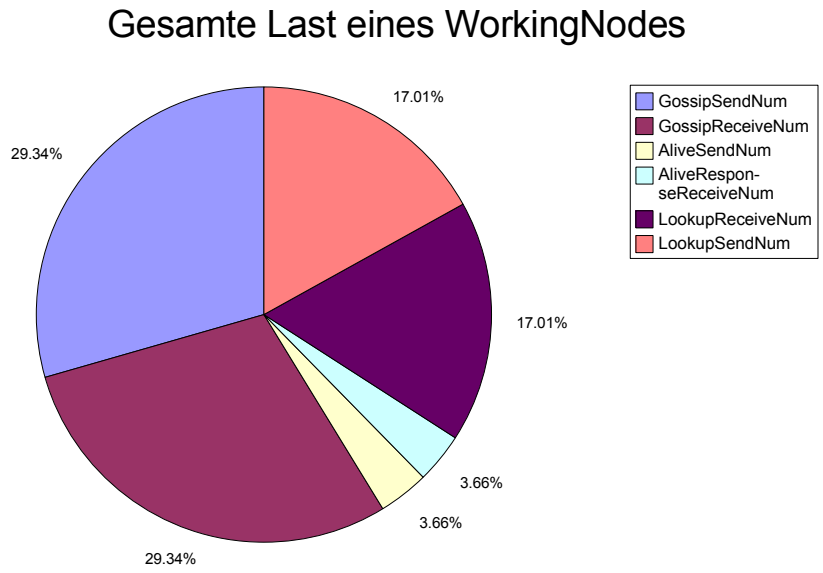

Abbildung 6.5: Verteilung der Arbeit eines Workingnode

wäre denkbar, ist jedoch für diese Arbeit unerheblich. Erheblich ist dafür allerdings die Last durch Gossip-Nachrichten, wenn man bedenkt einen zweiten Ring aus Working- und Churnnodes aufzubauen. Nicht nur, dass man die Last dadurch verdoppelt, es kommt zusätzliche Last durch Churnnodes hinzu. Diese Mehrarbeit muss im Kontrast zu den Verbesserungen gesehen werden.

In Abbildung 6.5 ist dargestellt wie sich die Gesamtlast pro einzelnem Knoten in diesem System durchschnittlich verteilt, wenn man einen Versuchsaufbau gemäß Tabelle 6.1 bzw. Tabelle 6.2 wählt. Wie zu erkennen ist, hat trotz einer sehr hohen, durch lookup-Nachrichten simulierten, Churnrate, der Anteil an Gossip-Nachrichten mit fast 60% der Nachrichtenlast überhand. Die Last, welche durch Hinzufügen von weiteren Knoten ins System abnehmen kann, ist der Anteil, der durch die lookup-Nachrichten gegeben ist.

6.4 Churn

6.4.1 Churn im Grundsystem

A priori ist noch nicht bekannt, wie die Anzahl der Workingnodes wächst oder schrumpft. Das wird in [8] erörtert. Geht man von einer gewissen Anzahl an Workingnodes als gegeben aus, so hat man ein statisches pub/sub-System. Alternativ kann dieses statische Szenario auch entstehen, wenn sehr viele Subscriptions und Unsubscriptions stattgefunden haben und immer nur eine, um eine feste Größe fluktuierende Anzahl an Subscribern, im System bleibt. Bei diesem Szenario bleiben stets gleichviel Workingnodes im System. Es soll nun ermittelt werden, wie sich das Grundsystem unter Churn verhält. Um ein solches Szenario aufzubauen wurden 300 Workingnodes als Grundlast nach und nach in das System gebracht. Sie sollten zufällig über zehn Topics verteilt arbeiten. Diese Topics werden als Topic null bis Topic neun benannt. Arbeit durch Churn wurde in diesem Szenario simuliert, indem jeder Knoten mit einer Wahrscheinlichkeit von 0.5 alle fünf Sekunden eine Suche bezüglich des Topics fünf mit zufälliger Topicorder gestartet hat. Die Simulation lief über einen simulierten Zeitraum von zwölf Stunden.

Abbildung 6.4.1 zeigt wie sich dabei die Arbeit, lookup-Nachrichten weiterleiten zu müssen, verteilt. Es entsteht ein Treppeneffekt, der aufzeigt, dass mit zunehmender Nähe zu Topic fünf, welches als einziges Churn hat, die Arbeit in den vorherigen Topics steigt. Die Differenz zwischen dem maximalen Fairnessquotienten, 0.7614212 für Topic fünf, und dem minimalen Fairnessquotienten, 0.0000008 für Topic vier, beträgt 0.7614204. Dies ist sehr unfair, da diese Differenz sehr groß ist. Je näher man dem Topic kommt, desto mehr lange Verbindungen reichen in das Topic und somit bekommen diese Knoten sehr häufig Arbeit. Theoretisch existieren Nachbarschaftlisten der Höhe 8 ($2^8 < 300 < 2^9$) und jedes Topic hat in etwa 30 Workingnodes. In jedes Topic reichen also $8 * 30 = 240$ Verbindungen aus rechten Nachbarschaftslisten bei insgesamt $8 * 300 = 2400$ rechten Nachbarschaftseinträgen im System. Das Problem dabei ist, dass zwar die Verbindungen gleichverteilt sind, nicht jedoch die Ziele des lookup. Wie schon früher gezeigt, müsste das Gebiet von Topic fünf um einiges größer sein, um die Arbeit fairer zu verteilen.

6.4.2 Auswirkung von Churnnodes

Was passiert nun bei dem gleichen Szenario, wenn im Topic fünf mehr Knoten als in den anderen Topics vorhanden sind? Um das zu testen wurden dem selben Setting 50, 100, 150 und 200 zusätzliche Knoten als Churnnodes hinzugefügt. Es wurden Versuche nach der gleichen Semantik wie im Grund-

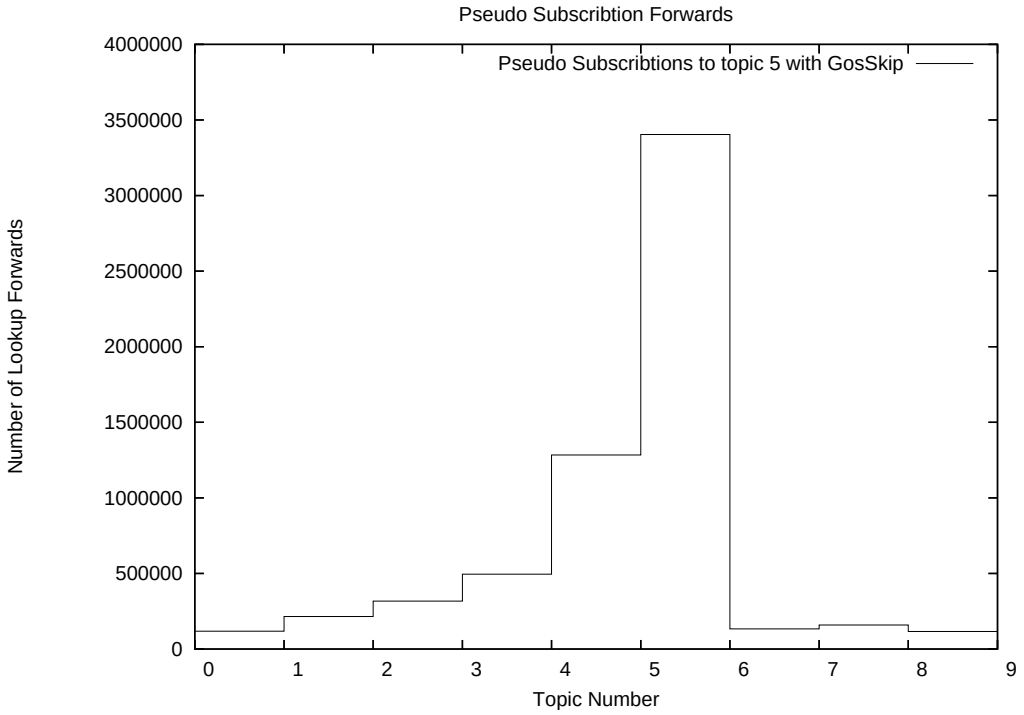

Abbildung 6.6: Lookup Nachrichten in einer Umgebung mit festen Workingnodes

system ohne Churnnodes durchgeführt.

Das Ergebnis dieser Untersuchung ist in Abbildung 6.7 zu sehen. Daneben wurde das Ergebnis mit 0 Churnnodes als Vergleich aufgelistet. Wie zu erwarten wurde mehr Arbeit in das Topic fünf verschoben. Die Differenz des maximalen und des minimalen Fairnessquotienten hat sich stetig von 0.76 im Grundsystem auf 0.46 bei 200 Churnnodes verbessert. Die Gesamtzahl der hops durch lookup-Nachrichten, welche das System während der Zeit belasteten, ist dabei allerdings von insgesamt 6365530 ohne Churnnodes auf 7002694 bei 200 Churnnodes gestiegen. Dies ist darin begründet, dass die durchschnittliche Hopzahl eines lookups bei 500 Knoten im System $\log_2(500) = 8,966$ beträgt, während sie sich bei 300 Knoten auf $\log_2(300) = 8,229$ beläuft, was sich über die Zeit aufsummiert. Hier ist nochmals zu erwähnen, dass bis zu 200 Knoten mehr im System Grundlast durch Gossip-Nachrichten produzierten.

Eine interessante Beobachtung scheint dabei zu sein, dass die Arbeit in zuvor schwer belasteten Topics logarithmisch abnimmt, während die Anzahl der Churnnodes linear ansteigt. Aus diesem Grund wurden noch die eben-

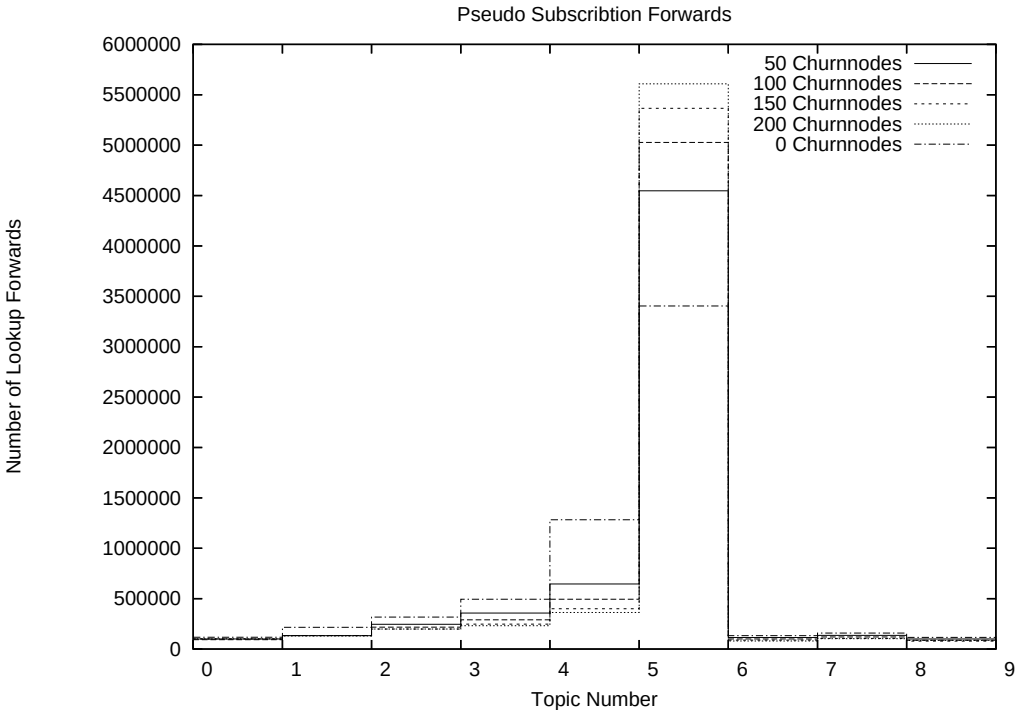

Abbildung 6.7: Verschiedene Mengen an Churnnodes

falls bestimmten Werte für 5, 10, 15, 20 und 25 Churnnodes zusammen mit den Werten der oben dargestellten Simulation verglichen. Das Ergebnis, wie sich die durch das Weiterleiten von lookup-Nachrichten entstehende Arbeit in Topic vier verändert, wenn mehr Churnnodes eingefügt werden, ist in Abbildung 6.8 linear interpoliert als Graph dargestellt. Dieses Ergebnis lässt vermuten, dass in einem Szenario ohne Churn in anderen Topics, ein sehr guter Wert für schwer belastete Topics nur sehr langsam erreicht wird.

Alle Ergebnisse bewegen sich im Rahmen der gleichen Anzahl an langen Verbindungen. Ob eine enorme Verbesserung durch den Übergang in eine Systemgröße mit 9 langen Verbindungen festzustellen ist, wurde im Folgenden getestet. Dass sich durch eine weitere Verbindung eine Verbesserung einstellt, ist in Abbildung 6.9 zu erkennen. Dort wurde eine Kurve aus Ergebnissen mit 200, 210, 215, 220 sowie 250 Churnnodes interpoliert. Dabei wurde die Gesamtgröße des Systems von 2^9 überschritten und es enstand eine weitere lange Verbindung. Ein Verfahren zu entwickeln, das darauf basiert soviele Knoten einzufügen, dass eine neue lange Verbindung entsteht, skaliert allerdings nicht mit der Systemgröße. Da lange Verbindungen logarithmisch mit der Systemgröße wachsen, müsste man exponentiell viele Knoten einfügen.

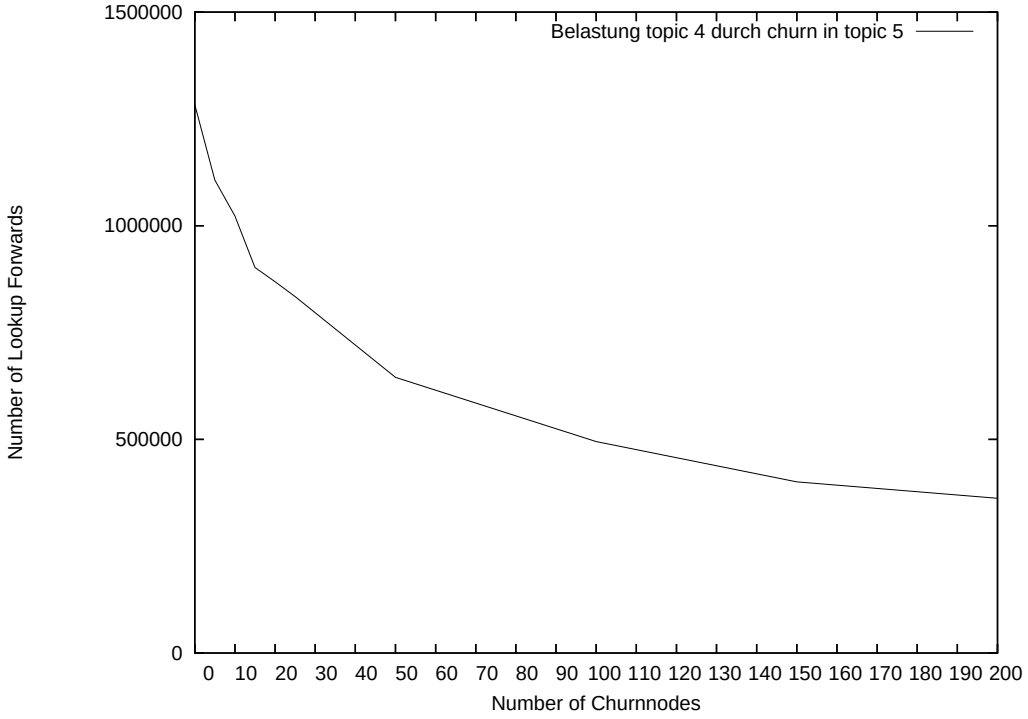

Abbildung 6.8: Das Verhalten von Topic vier für mehr Churnnodes

6.4.3 Behandlung von Churn durch Adaption

First come, first serve. Als erstes sollte der Adaptionsschritt, welcher als 'first come, first serve' bezeichnet wird, getestet werden. Wie im entsprechenden Kapitel erläutert, soll jeder Einstiegspunkt, also jeder Workingnode, nur begrenzt viele Churnnodes durch Pubnodes, welche sich bei ihm einschreiben, stellen lassen. Über die regelmäßigen Alive-Nachrichten teilte ein jeder Pubnode mit, wieviele Churnnodes er aktuell stellt. So wusste jeder Workingnode, wieviele Churnnodes durch, von ihm verwaltete Pubnodes, gestellt wurden. Das Grundsystem wurde aufgebaut, indem 200 Workingnodes innerhalb von 200 Sekunden dem System über zehn Topics gleichverteilt beitraten. Nachdem diese Knoten die Grundlage bildeten, wurde jede Sekunde ein Knoten erzeugt und bis zur Gesamtzahl von 2000 Knoten wurden Pubnodes, die dem Topic neun beitraten ins System gebracht. Ein jeder Workingnode beschränkte zu jedem Zeitpunkt das maximale Aufkommen an Churnnodes durch von ihm verwaltete Pubnodes auf eins. Damit wurde erreicht, dass sich das Gebiet des Topics in etwa verdoppelte. Diese Churnnodes blieben jeweils zehn Minuten im System. Das Experiment wurde beendet als keine Churnnodes mehr im System waren. In 6.10 ist das Ergebnis gegenüber dem selben

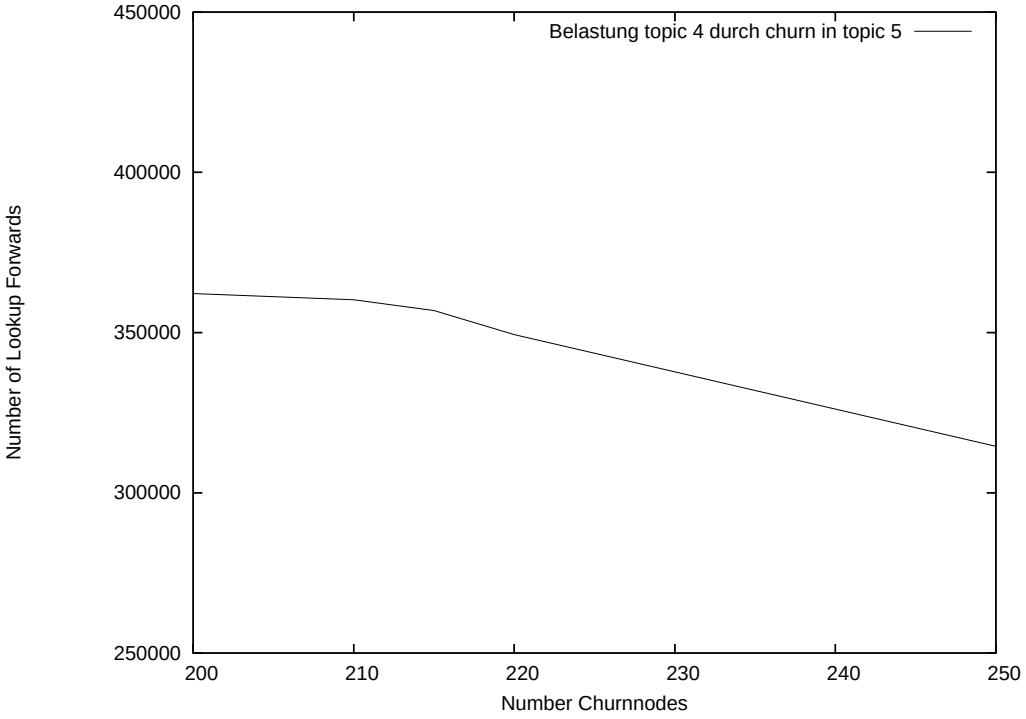

Abbildung 6.9: Das Verhalten von Topic vier mit einer weitern Langen Verbindung

Versuch ohne das Einfügen von Churnnodes abgegrenzt. Auch hier wurde die Last durch lookup- und join-Nachrichten pro Topic dargestellt. Es ist eine Entlastung von dem zuvor schwer belasteten Topic acht zu erkennen. Allerdings ist dies noch keine optimale faire Behandlung des Topics acht, denn sonst wäre die Last in der gleichen Größenordnung, wie die Last in Topic null bis sieben. Die Differenz der Fairnessquotienten von Topic neun und acht hat sich von 0.39 auf 0.29 verbessert.

Ein Fenster zur globalen Sicht. Der nächste Adaptionsschritt lief mit Anfragen, und wird im Kapitel Fenster zur globalen Sicht erläutert. Um diesen Ansatz zu testen wurde ein Grundsystem aufgebaut, in welchem 200 physikalische Knoten als Workingnodes dem System beitraten. Jeder physikalische Knoten, der einen Workingnode stellte, lies ebenso einen Pubnode dem System beitreten. Beide Arten an Knoten wurden exakt gleichverteilt über zehn Topics eingefügt. Somit hatte jedes Topic genau 20 Workingnodes und 20 Pubnodes. Nun wurde Churn verursacht, indem jeder Workingnode alle fünf Sekunden zufällig zehn Subscriptions durchführte. Auch diese benötigten lookup-Anfragen um Workingnodes zu finden. Diese Anfragen

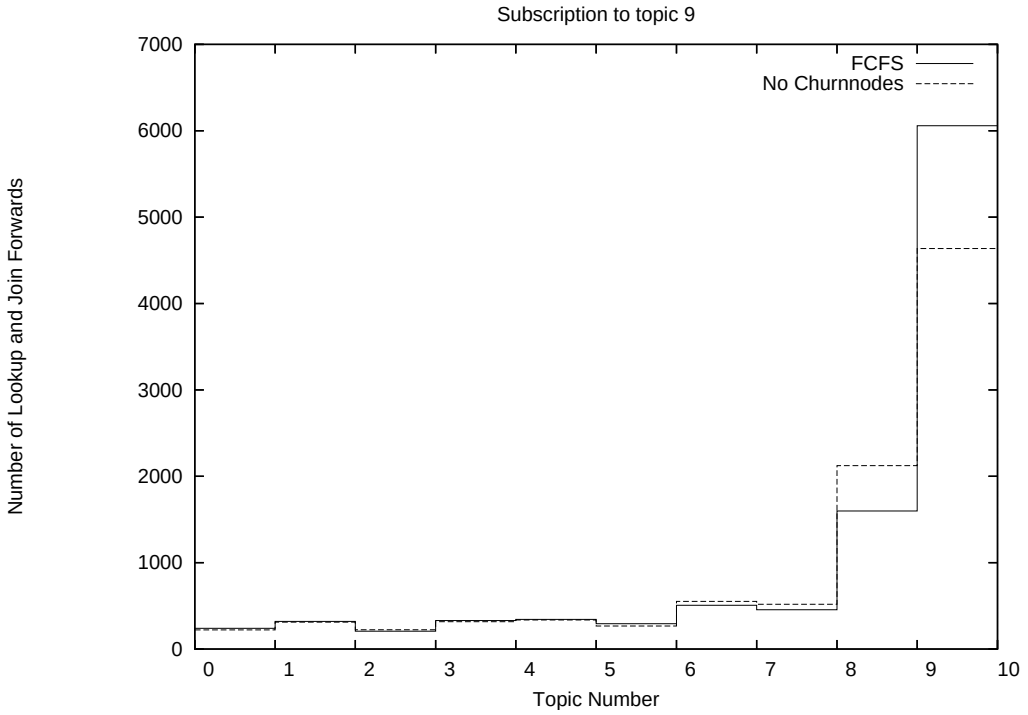

Abbildung 6.10: First come, first serve

wählten durch das BootstrapOracle zufällige Startpunkte für die Suche. Sie wurden von den entsprechenden verwaltenden Workingnodes nicht akzeptiert. Dieses Verhalten sollte also Churn durch Knoten, die sofort, nach dem sie dem Topic beigetreten waren, wieder eine Unsubscription durchführten, simulieren. Diese Subscriptions verteilten sich folgendermaßen auf die zehn Topics: 48% traten ausschließlich Topic fünf bei, 23% ausschließlich Topic zwei. Die restlichen 29% verteilten sich gleich über alle zehn Topics, was bedeutet, dass weitere 2,9% der Subscriptions Topic fünf und zwei belasteten. Anfragen und dementsprechend Adaptionsschritte wurden alle 120 Sekunden veranlasst. Mussten Knoten eingefügt werden, so hat ein Workingnode mit einer gewissen Wahrscheinlichkeit einen von ihm verwalteten Pubnode dazu beauftragt Workingnodes zu stellen. Mussten Knoten gehen, so entschied der Workingnode anhand einer Wahrscheinlichkeit, ob er selbst den GosSkip-Ring verlassen muss. Als Abgrenzung gegen die vorgenommen Adaptionsschritte ist eine zweite Simulation gestartet worden. Diese nahm die selbe Verteilung an Subscriptions vor, jedoch wurden keine Adaptionsschritte eingeleitet. Es wurden also weder Knoten hinzugefügt noch entfernt.

In Tabelle 6.3 ist das Verhalten des Systems dargestellt, wenn keine Adaption stattfindet. Für jedes der zehn Topics ist darin aufgeführt, wie sich

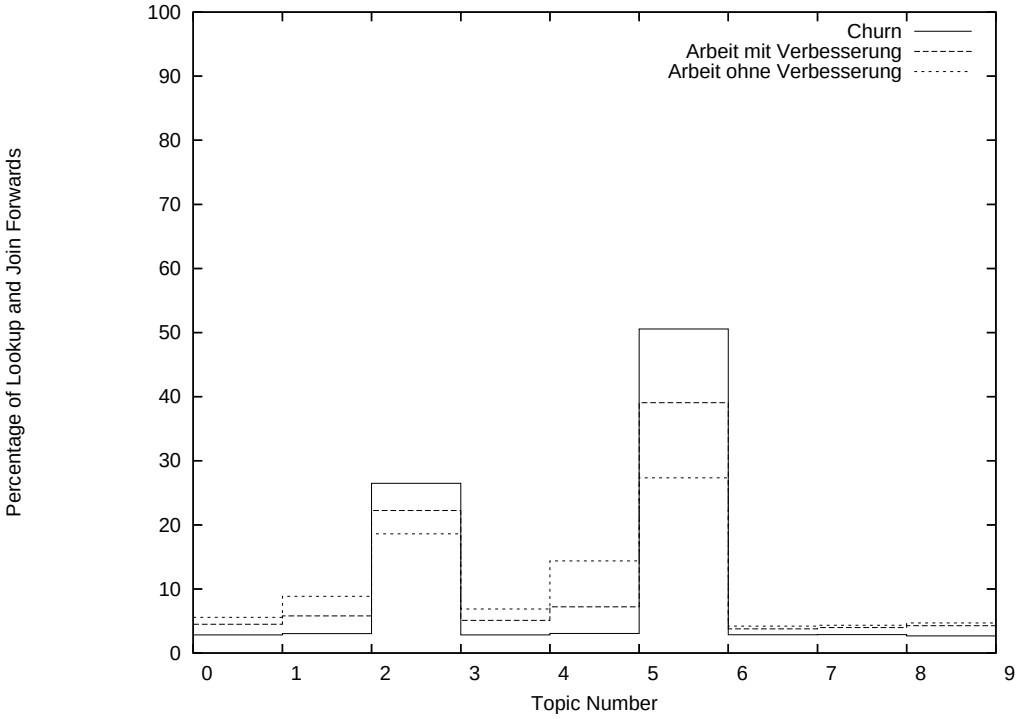

Abbildung 6.11: Anteil an Arbeit und Churn in %

der erwartete Churn verhält, wieviel Arbeit diese durch lookup-Nachrichten haben und wie sich der durch die hier beschriebene Metrik bestimmte Wert bezüglich des Fairnessquotienten verhält.

Die Ergebnisse der Untersuchung mit Adaptionsschritten sind in Tabelle 6.4 dargestellt. In der ersten Spalte der Tabelle ist das Topic, für welches die folgenden Werte ermittelt wurden, eingetragen. Spalte zwei zeigt den gemessenen Churn. Die dritte Spalte zeigt die Absolutwerte an lookup- und join-Weiterleitungen. Auch hier ist in der letzten Spalte die Fairness bezüglich Churn ermittelt worden.

Der Absolutwert für join- und lookup-Nachrichten schwankt stark. Dies liegt daran, dass bei der Adaption tendentiell eher Knoten dazugefügt wurden. So hat sich die Zahl der Workingnodes von anfänglichen 200 auf zuletzt etwa 700 erhöht. Durch diese Entwicklung stieg der Absolutwert an, da jeder Workingnode lookup-Nachrichten sendete. Dies ist nicht schlimm, da der prozentuale Anteil an Churn gleich blieb. Hierbei ist beim gemessenen Churn ein minimaler Fehler vorzufinden, da einjeder Workingnode ein Subscription-Gesuch als Rate an Churn in dem Topic, in welchem er arbeitet, gemessen hat. Da der letzte Knoten eines Vorgänger-Topics noch alle Subscriptions zu verwalten hat, die in das nächste Topic hineintreffen, aber eine kleinere

Topicorder als der erste Knoten in diesem Nachfolger-Topic besitzt, kommt dieser Fehler zu stande. Genaueres dazu ist im Abschnitt über Workingnodes beschrieben.

Anhand der Fairness kann man sehen, dass besonders die Topics mit hohem Churn von einem System ohne Adaption profitieren. Der Fairnessquotient von Churn zur Arbeit lookup-Anfragen weiterzuleiten ist sehr hoch und weicht stark von den Topics mit niedrigem Churn ab. Die maximale Differenz der Fairnessquotienten betrug dabei 0,36. Durch die angewandte Adaption konnte der Wert auf 0,16 gesenkt werden. Dies liegt näher am Optimum von 0 und ist demzufolge fairer.

Topic	Churn	Lookup	Fairness
0	501120	4505245	0.11
1	501120	7134720	0.07
2	4475520	14997309	0.3
3	501120	5543458	0.09
4	501120	11593763	0.04
5	8795520	22057760	0.4
6	501120	3384735	0.15
7	501120	3492156	0.14
8	501120	3796001	0.13
9	501120	4184960	0.12

Tabelle 6.3: Ergebnis ohne Adaption

Topic	Churn	Lookup/Join	Fairness
0	1527119	13782009	0.11
1	1641205	17779918	0.09
2	14290578	68315199	0.21
3	1530542	15656464	0.1
4	1651557	22139717	0.07
5	27280918	119986734	0.23
6	1545452	11618970	0.13
7	1554013	12233409	0.13
8	1444741	13101255	0.11
9	1493162	12540978	0.12

Tabelle 6.4: Ergebnis mit Adaption

Da bei diesem Adaptionsschritt der prozentuale Anteil an Churn auf den prozentualen Anteil an Knoten im System angepasst werden sollte, ist die

prozentuale Verteilung der Arbeit und des Churns auf die Topics in Abbildung 6.11 dargestellt. Wie man sehen kann, nähert sich die Kurve, welche Arbeit durch lookup- und join-Weiterleitung symbolisiert, stärker dem eigentlichen Churn-Prozentsatz an.

Kapitel 7

Schlussfolgerung und Ausblick

Churn ist in pub/sub-Systemen als die Rate an Subscriptions und Unsubscriptions definiert. In dieser Arbeit wurde gezeigt, dass diese zusätzliche Arbeit in weniger belasteten Topics als dem, mit hohen Churn, vorallem durch Subscriptions gegeben ist. Negativ wirkt sich dabei vorallem die Suche nach einem Einstiegspunkt für ein bestimmtes Topic auf andere aus. Fairness ist die Verteilung der Arbeit anhand des Nutzens. Das Fairnesskriterium bezüglich Churn ist durch die Verteilung von Arbeit auf Topics proportional zum Churn definiert. Suchanfragen weiterleiten zu müssen ist hierbei die zu verteilende Arbeit. In vielen Systemen [6, 7, 5] ist eine unfaire Behandlung von Teilnehmern des Systems durch unterschiedlichen Churn in verschiedensten Topics gegeben.

Das hier vorgestellte System bietet die Möglichkeit dynamisch Arbeit zu verteilen. Deshalb ist hier die Möglichkeit gegeben Arbeit, insbesondere Arbeit Suchanfragen weiterzuleiten, dynamisch anhand des Churns auf die Topics zu verteilen. Das System baut auf einem P2P-Overlay namens GosSkip auf. Damit Arbeit verteilt werden kann, wurde eine Trennung der Arbeit und des Nutzens durch Working-, Churn- und Pubnodes vorgenommen. Working- und Churnnodes sind Arbeitseinheiten, welche die selbe Arbeit leisten. Die Anzahl der Knoten wird dynamisch und adaptiv angepasst. Pubnodes in Topics mit hohem Churn stellen mehr Arbeitseinheiten, als Pubnodes in Topics mit geringerem Churn. Das System kann diese Dynamik bewältigen. Dementsprechend passt es sich an jede Systemgröße an. Dazu ist ein adaptiver Mechanismus zur Ausfallsicherheit, sowie einer zur Anpassung der Größe einer Nachbarschaftsliste, vorhanden.

Damit Churn behandelt werden kann, sind zwei Verfahren vorgestellt worden. Beim ersten Verfahren wird lokales Wissen, welches ein Workingnode nur durch Aufsummieren der bei ihm eingegangenen Subscription-Gesuche und Unsubscription-Gesuche erhält, verwendet. Mit diesem Wissen kann der ein-

zelne Workingnode lokal entscheiden, ob Maßnahmen ergriffen werden sollten, um Arbeit zu verteilen. Ein zweiter Ansatz versuchte, durch stichprobenhafte Anfragen von zufälligen Knoten, eine globale Sicht anzunähern. Anhand dieser Informationen entschied ein Workingnode, ob das Gebiet vergrößert oder verkleinert werden sollte. Durch Evaluationen dieser Ansätze konnte gezeigt werden, dass insbesondere der zweite Ansatz das Verhältnis von Churn in Topics zur entsprechenden Arbeit Suchanfragen weiterzuleiten verbessert hat.

Grundsätzlich ist die Belastung unter Churn, die durch die Suche nach einem Einstiegspunkt in einen Bereich für ein Topic entsteht, durch die hier vorgestellten Methoden und den hier vorgestellten Systemaufbau einschränkbar. Die Arbeit komplett aus nicht betroffenen Topics zu nehmen ist schwer möglich. Dazu müsste ein Verfahren in zugesicherten O(1) das Topic erreichen oder eine komplette globale Sicht erhalten.

Evaluierungen haben gezeigt, dass mit den vorgestellten Algorithmen und dem vorgestellten Systemaufbau ein großer Schritt in Richtung Fairness unter Churn unternommen wurde. Sie zeigten aber auch, dass weitere Schritte in diese Richtung unternommen werden müssen, um eine optimale Fairness zu erreichen.

Der vorgestellte Algorithmus, in dem durch Anfragen eine angenäherte globale Sicht ermöglicht wurde, baut darauf, dass er einen repräsentativen Schnitt der Topics durch Anfragen zufällig erreicht. Je mehr Topics vorhanden, umso schlechter werden allerdings die Stichproben. Um diesem Problem entgegen zu wirken wäre es sinnvoll, das Wissen des physikalischen Knoten auszunützen. Es wird davon ausgegangen, dass jeder physikalische Knoten mehrere virtuelle Knoten ins System eingebracht hat. Jeder virtuelle Knoten könnte das durch Stichproben gesammelte Wissen anderer virtueller Knoten auf dem physikalischen Knoten dazu verwenden, um seine Stichproben zu verbessern und damit eine bessere angenäherte globale Sicht zu erhalten.

Möchte man nicht die Workingnodes durch das Wissen aus der angenäherten globalen Sicht verteilen, sondern Churnnodes, so muss man den Algorithmus modifizieren. Man könnte durch die langen Verbindungen des Workingnoderinges eine Gesamtzahl an zu vergebenden Churnnodes bestimmen, welche in Höhe der Größe des Systems läge. Mit dem Algorithmus zur angenäherten globalen Sicht kann ein jeder Knoten bestimmen, wieviel Prozent dieser Gesamtzahl ein Topic stellen müsste und dementsprechend Churnnodes einfügen.

Alternative und zusätzliche Überlegungen zur variablen Größe von Bereichen könnten in Betracht ziehen, ein zweidimensionales lookup anzustreben. Hierbei könnte so schnell wie möglich eine lange Verbindung gesucht werden, die in das Gebiet des Topics führt und von dort aus die Position des entspre-

chenden Knotens innerhalb des Gebietes gesucht werden. Diese Möglichkeit führt aller Wahrscheinlichkeit nach zu einer erhöhten Hopzahl bei der Suche, aber zu einer Entlastung der Gebiete vor dem durch Churn belasteten Gebiet. Die Kosten durch zusätzliche hops müssen gegenüber der Entlastung aufgerechnet werden.

Das hier vorgestellte pub/sub-System kann noch weiter verbessert werden. So sollte es in Hinblick auf Zuverlässigkeit der Ereignisauslieferung verbessert werden. Die Einschränkung, dass ein Pubnode nach dem Ausfall seines Workingnodes erneut seinem Topic beitreten muss sollte deswegen behoben werden. Entsprechende Lösungsansätze könnten hierbei sein, dass jeder Pubnode redundant mögliche Workingnodes hält. Diese bekommt er dann eventuell von seinem Verwalter durch Alive-Nachrichten gesendet. Alternativ könnte ein Workingnode, sobald er einen Ausfall seines linken Nachbarn bemerkt, die von diesem verwalteten Pubnodes übernehmen. Dazu müssten ebenfalls redundant Informationen gehalten werden, dieses mal allerdings bei dem Workingnode.

Ein Aspekt, welcher nicht bei dem hier vorgestellten System betrachtet wurde, ist die effektive Ereignisauslieferung. Für die vorgestellten Untersuchungen war diese nicht notwendig. Ineffizient geschieht diese durch Fluten innerhalb einer doppelt verketteten Liste und damit in O(t) Zeiteinheiten, sei $t \in \mathbb{N}$ die Anzahl der Knoten innerhalb eines Topics. Es wäre effektiver und zuverlässiger einen Multicastbaum oder eine ähnliche Struktur zu halten, wobei hier die Schwierigkeit darin besteht, dass die Pubnodes zum Aufbau und Erhalt einer solchen Struktur nicht involviert werden sollten.

Das hier vorgestellte Prinzip, um Fairness bereitzustellen, kann auf andere pub/sub-Systeme, welche durch Veränderung der Größe eines Topics Arbeit verteilen, angewandt werden. TERA [7] stellt ein alternatives pub/sub-System dar, welches offensichtlich ähnliche Eigenschaften wie GosSkip durch die Veränderung von Bereichsgrößen zeigt. Größere Bereiche versenden häufiger Advertisments. Ebenso senden mehr virtuelle Knoten Advertisments, und somit ist ein großer Bereich in mehr APT s vertreten. Durch bessere Repräsentation in APT s würde das Gebiet häufiger in O(1) gefunden werden. Würde man das Konzept von Workingnodes, Churnnodes und Pubnodes auf TERA anwenden und zugleich das Topic-Overlay anders gestalten, so könnte man die hier vorgestellten Methoden in Variation dort anwenden. Workingnodes sowie Churnnodes wären im P2P-Overlay, Pubnodes im Topic-Overlay angesiedelt. Durch stabile Knoten, also Knoten die schon länger im System sind, als Workingnodes und damit als Einstiegspunkte in das Topic, sowie durch Vergrößerung der Gebiete mit Hilfe der Churnnodes, ebenfalls von stabileren Knoten, würde sich ebenfalls ein äquivalentes Verhalten unter Churn wie bei dem hier vorgestellten System ergeben. Paralelle

Untersuchungen an diesem System könnten zeigen, unter welcher Parametrisierung das hier vorgestellte System bzw. TERA sich besser verhält.

Literaturverzeichnis

[1] William Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures*, pages 437–449, 1989.

[2] R. Guerraoui, S. Handurukande, K. Huguenin, A.-M. Kermarrec, F. Le Fessant, and E. Riviere. GosSkip, an Efficient, Fault-Tolerant and Self Organizing Overlay Using Gossip-based Construction and Skip-Lists principles. In *Proceedings of the IEEE International Conference on Peer-to-Peer Computing*, 2006.

[3] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. OverSim: A Flexible Overlay Network Simulation Framework. In *Proceedings of 10th IEEE Global Internet Symposium (GI '07) in conjunction with IEEE INFOCOM 2007, Anchorage, AK, USA*, May 2007.

[4] Hongzhou Liu, Venugopalan Ramasubramanian, and Emin Gün Sirer. Client behavior and feed characteristics of rss, a publish-subscribe system for web micronews. In *Proceedings of the Internet Measurment Conference*, pages 29–34. USENIX Association, 2005.

[5] Gregory Chockler, Roie Melamed, Yoav Tock, and Roman Vitenberg. Spidercast: a scalable interest-aware overlay for topic-based pub/sub communication. In *Proceedings of the 2007 inaugural international conference on Distributed event-based systems (DEBS '07)*, pages 14–25, New York, NY, USA, 2007. ACM Press.

[6] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication*, pages 30–43, 2001.

[7] Roberto Baldoni, Roberto Beraldi, Vivien Quema, Leonardo Querzoni, and Sara Tucci-Piergiovanni. Tera: topic-based event routing for peer-to-peer architectures. In *Proceedings of the 2007 inaugural international conference on Distributed event-based systems (DEBS '07)*, pages 2–13, New York, NY, USA, 2007. ACM Press.

[8] Simon Gansel. Faire Arbeitsteilung in dynamischen Publish/Subscribe Systemen. Study Thesis, University of Stuttgart, 2007.

[9] S. Bhola, R. Strom, S. Bagchi, Y. Zhao, and J. Auerbach. Exactly-once delivery in a content-based publish-subscribe system. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'2002)*, 2002.

[10] Arnas Kupsys, Stefan Pleisch, Schiper Schiper, and Matthias Wiesmann. Towards JMS Compliant Group Communication - A Semantic Mapping. In *Proceedings of the Network Computing and Applications (NCA'04)*, pages 131–140, Washington, DC, USA, 2004. IEEE Computer Society.

[11] Andrew S. Tannenbaum and Marten van Steen. *Distributed Systems - Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 4th edition, 2002.

[12] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[13] Eytan Adar and Bernardo A. Huberman. Free riding on gnutella. *First Monday*, September 2000.

[14] Bram Cohen. Incentives build robustness in bittorrent, 2003.

[15] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[16] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.

[17] Manfred Hauswirth and Schahram Dustdar. Peer-to-Peer: Grundlagen und Architektur. *Datenbank-Spektrum*, 5(13):5–13, 2005.

[18] Quin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks, 2001.

[19] Patrick Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, and Laurent Massoulié. From epidemics to distributed computing. *IEEE Computer*, 37(5):60–67, May 2004.

[20] R. Guerraoui, S. B. Handurukande, and A.-M. Kermarrec. GosSkip: a Gossip-based Structured Overlay Network for Efficient Content-based Filtering. Technical report, 2004.

[21] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the International Conference on Distributed Systems Platforms (IFIP/ACM)*, pages 329–350, November 2001.

[22] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scalable application-level anycast for highly dynamic groups. pages 47–57. 2003.

[23] T. Euler. Churn prediction in telecommunications using miningmart. In *Proceedings of the Workshop on Data Mining and Business (DMBiz)*, 2005.

[24] Sébastien Baehni, Rachid Guerraoui, Boris Koldehofe, and Maxime Monod. Towards Fair Event Dissemination. In *Proceedings of the 27th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW 2007)*, 2007.

[25] S. Naicken, A. Basu, B. Livingston, and S. Rodhetbhai. A Survey of Peer-to-Peer Network Simulators. In *Proceedings of The Seventh Annual Postgraduate Symposium*, 2006.

[26] Andras Varga. The omnet++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM'2001)*, 2001.

[27] http://www.omnetpp.org, October 2007.

Erklärung

Hiermit versichere ich, diese Arbeit
selbständig verfasst und nur die
angegebenen Quellen benutzt zu haben.

(Beate Ottenwälder)