



Universität Paderborn  
Fakultät für  
Elektrotechnik, Mathematik und Informatik

---

Seminar: Perlen der theoretischen Informatik

## Skip Graphen

Autor

Jürgen Brinkmann, [ilak@upb.de](mailto:ilak@upb.de)

16.01.2005

.....

# Inhaltsverzeichnis

Kapitel 1	Einleitung	1
.....		
Kapitel 2	Skip Graphen	2
.....		
2.1	Skip Listen	2
2.2	Struktur der Skip Graphen	2
2.2.1	Formale Definition	3
.....		
Kapitel 3	Algorithmen für Skip Graphen	5
.....		
3.1	Die search Operation	5
3.2	Die insert Operation	5
.....		
Kapitel 4	Reparaturmechanismen	6
.....		
4.1	Die Invariante aufrecht erhalten	6
4.2	Die Skip Graph Bedingungen wieder herstellen	7
.....		
Kapitel 5	Fehlertoleranz	10
.....		
5.1	Ungünstige Fehler	10
5.2	Zufällige Fehler	10
.....		
Kapitel 6	Lastenausgleich	11
.....		
Kapitel 7	Fazit	12
.....		
Literaturverzeichnis		13
.....		

# Kapitel 1

---

## Einleitung

Peer-to-Peer Netzwerke sind verteilte Systeme, die keine zentrale Autorität besitzen, welche für das effiziente Bestimmen verteilter Ressourcen zuständig ist. Solche Systeme sind in kurzer Zeit für Internetanwendungen sehr populär geworden. Eine nähere Betrachtung über aktuelle Peer-to-Peer Systeme zeigt die gewünschten Fähigkeiten, die ein solches System haben sollte. Diese Fähigkeiten sind unter Anderem:

- Dezentralisierung
- Skalierbarkeit
- Datenverfügbarkeit
- Lastenausgleich
- Dynamisches Hinzufügen und Löschen von Peer-Knoten
- Effiziente und komplexe Suche
- Berücksichtigung der Geographie bei der Suche
- Ausnutzung von räumlicher und zeitlicher Lage bei der Suche

Heutige Peer-to-Peer Systeme wie CAN [RFH<sup>+</sup>01] and Chord [SMK<sup>+</sup>01] benutzen den „Distributed Hash Table“ (DHT) Ansatz um die Skalierbarkeit zu gewährleisten. Dabei zerkleinern (engl.: to hash) sie den Schlüssel einer Ressource, um zu bestimmen welcher Knoten gespeichert wird. So verteilen sie die Lasten der Knoten im Netzwerk. Die Hauptaufgabe in solchen Systemen ist es den Knoten zu bestimmen, der die gewünschte Ressource speichert. Zu diesem Zweck existiert ein Überlagerungsgraph, in welchem die Stelle und die Ressource der Knoten durch die Hash-Werte bestimmt werden kann.

Die Struktur von Systemen wie Chord und CAN gleicht einem ausgewogenem Baum in dem das Halten des Gleichgewichts von einer Fastgleichverteilung der Hashfunktion abhängt. Da die Schlüssel gehasht sind, birgt diese Datenstruktur auch nur Hash Table Funktionalitäten und berücksichtigen keine lokalen Eigenschaften. Daher ist eine solche Struktur nicht robust gegenüber ungünstigen Fehlern und stabilisiert sich nicht selbst.

Der von [AS03] vorgestellte Ansatz nutzt die unterliegende Treestruktur aus, um die Funktionalitäten eines Trees zu gewährleisten. Der so entstandene Skip Graph Ansatz unterstützt den Lastenausgleich im Netzwerk. In dieser Struktur kann das Lokalisieren von Ressourcen und das Hinzufügen und Löschen von Knoten in logarithmischer Zeit geschehen. Dabei benötigt jeder Knoten im Skip Graph nur logarithmischen Speicherplatz um Informationen über seine Nachbarn zu speichern.

## Kapitel 2

# Skip Graphen

### 2.1 Skip Listen

Eine Skip Liste [Pug90] ist eine zufällige, ausgewogene Tree-Datenstruktur, die man sich vereinfacht als ein Turm von steigenden, verstreuten Link-Listen vorstellen kann. Level 0 einer Skip Liste ist eine verknüpfte Liste mit existierenden allen Knoten, sortiert in aufsteigender Reihenfolge nach dem Schlüssel der Knoten. Für jedes Level  $i$ , das größer als 0 ist, gilt für einen Knoten im Level  $i - 1$ , dass er im Level  $i$  mit einer bestimmten Wahrscheinlichkeit  $p$  erscheint.

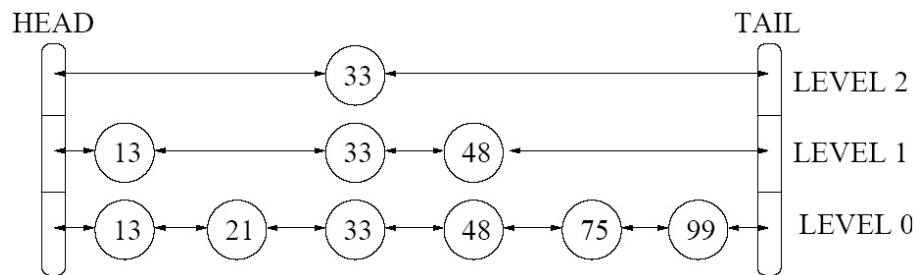


Abbildung 2.1: Eine Skip Liste (©[AS03])

### 2.2 Struktur der Skip Graphen

In [AS03] wird eine Datenstruktur vorgestellt, die ähnlich wie bei den Skip Listen typische Binätree-Operationen unterstützt. Die Elemente der Datenstruktur sind in separaten Orten in einem stark verteilten System gespeichert, daher kann es oft zu unvorhersehbaren Fehlern kommen. Der vorgestellten Datenstruktur gelingt es solche Fehler abzufangen.

Eine Skip Liste reicht für die Zwecke eines Peer-to-Peer-Netzwerkes laut [AS03] nicht aus, da sie auf Grund ihrer verteilten Organisation oft redundant gespeichert wird. Daher ist sie verwundbar für Fehler und Auseinandersetzungen mit anderen Peers. Außerdem führt die Tatsache, dass nur ein paar Knoten in dem höchsten Level der Liste zu finden sind, dazu, dass jeder dieser Knoten ein möglicher Fehlerpunkt ist. Denn durch dessen Entfernen teilt sich die Liste. Jeder Knoten in den höheren Leveln einer Skip Liste ist daher ein Krisenherd, der einen konstanten Teil der Such-Operationen bearbeiten muss. Skip Listen können außerdem nicht garantieren, dass einzelne Knoten von den übrigen Knoten getrennt werden, selbst wenn nur wenigen zufällige Fehler auftreten. Da jeder

Knoten im Durchschnitt zu nur  $O(1)$  anderen Knoten verbunden ist, wird sogar bei einer konstanten Zufälligkeit von Knotenfehlern ein großer Teil der übrigen Knoten isoliert.

Die Lösung, die in [AS03] vorgestellt wird, ist eine Generalisierung von den Skip Listen ([Pug90]), den Skip Graphen. So wie bei den Skip Listen, ist bei den Skip Graphen jeder Knoten ein Mitglied von mehreren verknüpften Listen. Die Liste im Level 0 beinhaltet alle Knoten der Sequenz. Skip Graphen unterscheiden sich von Skip Listen dadurch, dass es mehrere Listen im Level  $i$  geben kann und dass jeder Knoten in einer der Listen im Level  $i$  zu finden ist, bis die Knoten nach durchschnittlich  $O(\log n)$  Levels in einzelne Knoten geteilt werden.

Da es mehrere Listen in jedem Level geben kann, ist die Chance, dass ein einzelner Knoten an einer Suche beteiligt ist klein. Dadurch werden einzelne Fehlerpunkte und Krisenherde behoben. Des Weiteren hat jeder Knoten durchschnittlich  $\Theta(\log n)$  Nachbarn, und mit hoher Wahrscheinlichkeit ist kein Knoten isoliert ([AS03]).

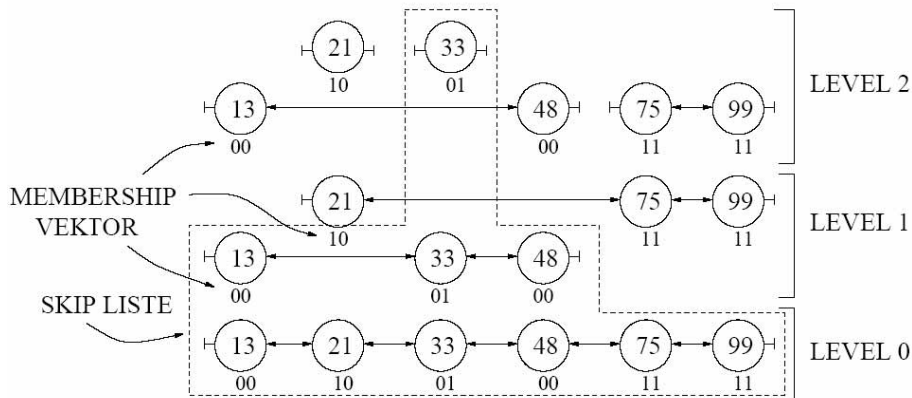


Abbildung 2.2: Ein Skip Graph mit  $\lceil \log N \rceil = 3$  Levels (©[AS03])

Um über die Fehlertoleranz hinaus eine Suche in der Zeit  $O(\log n)$  zu unterstützen, scheint für eine verteilte Datenstruktur, basierend auf Knoten in einem eindimensionalen Raum, die zufällig verbunden sind, ein Ausmaß von  $\Omega(\log n)$  notwendig zu sein ([ADS02]).

### 2.2.1 Formale Definition

Welchen Listen ein Element  $x$  angehört wird von dem so genannten **Membership Vektor**  $m(x)$  kontrolliert.  $m(x)$  wird als unendliches zufälliges Wort über einem bestimmten Alphabet gesehen, obwohl in der Praxis nur ein Präfix von  $m(x)$  der durchschnittlichen Länge  $O(\log n)$  generiert werden muss. Die Idee des Membership Vektor ist, dass jede doppelt verknüpfte Liste in dem Skip Graph mit einem endlichen Wort  $\omega$  gekennzeichnet ist und ein Element  $x$  sich in der mit  $\omega$  gekennzeichneten Liste nur dann befindet, wenn  $\omega$  ein Präfix von  $m(x)$  ist. Wie in Abbildung 2.2 zu sehen ist, gibt es im Level 0 nur eine Liste. Diese Liste wird durch  $\omega = \epsilon$ , dem leeren Wort, gekennzeichnet. Im Level 1 existieren zwei Listen. In der einen, mit  $\omega = 0$  gekennzeichneten Liste, befinden

sich alle Knoten, die die Zahl Null als Präfix haben. In der anderen Liste, die mit  $\omega = 1$  gekennzeichnet ist, finden sich alle Knoten wieder, die die Zahl Eins als Präfix haben. In den vier Listen im Level 2 befinden sich Knoten mit Membership Vektoren aus der Menge  $\{00, 01, 10, 11\}$ . Jedes Element aus dieser Menge kennzeichnet eine Liste im Level 2.

Um diese Struktur diskutieren zu können, wird folgende Notation benötigt: Sei  $\Sigma$  ein endliches Alphabet, sei  $\Sigma^*$  die Menge der endlichen Worte, die aus den Buchstaben aus  $\Sigma$  besteht, und sei  $\Sigma^\omega$  die Menge aus allen unendlichen Worten. Es werden Indizes benutzt, um auf einzelne Buchstaben eines Wortes zu verweisen, beginnend mit dem Index 0. Ein Wort  $\omega$  ist gleich zu  $\omega_0\omega_1\omega_2 \dots$ . Sei  $|\omega|$  die Länge von  $\omega$ , mit  $|\omega| = \infty$ , wenn  $\omega \in \Sigma^\omega$ . Wenn  $|\omega| \geq i$  ist, bezeichnet  $\omega \upharpoonright i$  das Präfix von  $\omega$  mit der Länge  $i$ .

Bei den Skip Graphen ist das unterste Level immer eine doppelt verknüpfte Liste  $S_\epsilon$ , die alle Elemente sortiert enthält. Im Allgemeinen enthält die doppelt verknüpfte Liste  $S_\omega$ , für jedes  $\omega \in \Sigma^*$ , alle  $x$  in ansteigender Reihenfolge, für die  $\omega$  ein Präfix von  $m(x)$  ist. Eine einzelne Liste  $S_\omega$  ist Teil vom Level  $i$ , wenn  $|\omega| = i$ .

Ein Skip Graph kann des Weiteren als ein zufälliger Graph gesehen werden, in dem es eine Kante zwischen  $x$  und  $y$  gibt, wenn  $x$  und  $y$  in irgend einem  $S_\omega$  an einander an liegen.  $x$ 's linke und rechte Nachbarn im Level  $i$  sind dann als seinen unmittelbaren Vorgänger und Nachfolger definiert. Beziehungsweise  $S_{m(x)\upharpoonright i}$ , oder  $\perp$ , wenn so ein Knoten nicht existiert. Im Folgenden wird  $xL_i$  als  $x$ 's linker und  $xR_i$  als  $x$ 's rechter Nachbar im Level  $i$  bezeichnet. Im Allgemeinen werden  $L_i$  und  $R_i$  als stilisierte Operatoren benutzt, um Ausdrücke wie  $xR_iR_{i-1}^2$  zu erlauben.

Ein alternativer Blickwinkel auf einen Skip Graphen ist ein Tree bestehend aus mehreren Skip Listen, die ihre unteren Level teilen. Wenn man eine Skip Liste formell als zufällige Variablen  $S_0, S_1, S_2, \dots$  betrachtet, in denen der Wert von  $S_i$  die Liste auf dem Level  $i$  ist, dann gilt:

**Lemma 2.1** *Sei  $\{S_\omega\}$  ein Skip Graph mit dem Alphabet  $\Sigma$ . Dann gilt: für jedes  $z \in \Sigma^\omega$  ist die Sequenz  $S_0, S_1, S_2, \dots, S_n$ , bei der jedes  $S_i = S_{z\upharpoonright i}$  ist, eine Skip Liste mit  $p = |\Sigma|^{-1}$ . ([AS03])*

*Beweis:* Über Induktion von  $i$ . Die Liste  $S_0$  ist gleich  $S_\epsilon$ , welche die Basisliste aller Elemente ist. Ein Element  $x$  befindet sich in  $S_i$ , wenn  $m(x) \upharpoonright i = z \upharpoonright i$  ist. Aus dieser Bedingung folgt: Die Wahrscheinlichkeit, dass sich  $x$  außerdem in  $S_{i+1}$  befindet, ist  $m(x)_{i+1} = z_{i+1}$ . Dieses Ereignis tritt mit der Wahrscheinlichkeit  $p = |\Sigma|^{-1}$  auf. Es ist leicht zu erkennen, dass es unabhängig zu den korrespondierenden Ereignissen für jedes andere  $x'$  in  $S_i$  ist. Daher befindet sich jedes Element, dass sich in  $S_i$  befindet auch in  $S_{i+1}$  mit unabhängiger Wahrscheinlichkeit  $p$  und  $S_0, S_1, S_2, \dots, S_n$  bilden eine Skip Liste. ■

In einem Peer-to-Peer System ist jede Quelle ein Knoten in einem Skip Graphen und die Knoten sind bezüglich des zugehörigen Schlüssels sortiert. Jeder Knoten speichert die Adresse und die Schlüssel seiner beiden Nachbarn auf jedem der  $O(\log n)$  Level. Außerdem benötigt jeder Knoten  $O(\log n)$  bits als Raum für seinen Membership Vektor ([AS03]).

## Kapitel 3

# Algorithmen für Skip Graphen

In diesem Kapitel werden die *search* und die *insert* Operationen für Skip Graphen beschrieben. Die Beschreibung der *delete* Operation wird ausgelassen, da sie einfach aus den vorgestellten Operationen abgeleitet werden kann.

### 3.1 Die search Operation

Die search Operation ist identisch mit der search Operation der Skip Listen, mit dem Unterschied, dass man den Algorithmus auf einem verteilten System laufen lassen kann. Die Suche beginnt im obersten Level eines Knoten, der einen Schlüssel sucht. Über dieses gesamte Level wird gegangen, ohne den gesuchten Schlüssel zu überschreiten. Wird ein Knoten gefunden, der einen Zeiger besitzt, der auf einen Schlüssel zeigt, der vor (hinter) dem gesuchten Schlüssel liegt wird das nächst tiefere Level durchsucht, bis man das Level 0 erreicht hat. Entweder die Adresse des Knoten, der den gesuchten Schlüssel speichert (wenn er existiert), oder die Adresse des Knoten, der den Schlüssel beinhaltet, der am nächsten zum benötigten Schlüssel liegt, wird zurückgegeben. Der Algorithmus ist in [AS03] zu finden.

Nach [AS03] benötigt die search Operation in einem Skip Graphen mit  $n$  Knoten  $O(\log n)$  Zeit und  $O(\log n)$  Nachrichten.

### 3.2 Die insert Operation

Ein neuer Knoten  $n'$  fügt sich selbst in eine Liste in jedem Level ein bis er sich selbst alleine in einer Liste in irgendeinem Level befindet. Beim Level 0 wird sich  $n'$  mit einem Knoten verbinden, der am nächsten zu seinem eigenen Schlüssel ist. Bei jedem Level  $i$ ,  $i \geq 1$ , wird  $n'$  versuchen den nächsten Knoten  $x$  im Level  $i - 1$  zu finden, für den gilt  $m(x) \uparrow i = m(n') \uparrow i$ . Mit diesem Knoten  $x$  verbindet sich  $n'$  im Level  $i$ . Jeder existierende Knoten kann das Bestimmen von  $m(x)$  verhindern, bis ein neuer Knoten erscheint und den Wert des existierenden Knotens wissen will. Dadurch wird erreicht, dass zu jeder Zeit nur eine endliche Anzahl von Präfixen, für jeden Membership, generiert werden muss.

Das Einfügen kann schwieriger sein, wenn es gleichzeitig zu Knotenverknüpfungen kommt. Bevor sich  $n'$  mit seinen Nachbarn verbindet, muss er erst untersuchen, ob seine Verknüpfung nicht die Skip Graph Eigenschaften verletzt. Wenn sich also ein beliebiger neuer Knoten mit dem Knoten  $n'$  und seinen vorher festgelegten Nachbarn verbinden will, muss  $n'$ , falls nötig, über die neuen Knoten gehen, bevor er sich an dem richtigen Platz einfügt. Der Pseudo-Code des Algorithmus findet sich in [AS03].

Laut [AS03] benötigt die insert Operation in einem Skip Graph mit  $n$  Knoten  $O(\log n)$  Zeit und  $O(\log n)$  Nachrichten.

## Kapitel 4

# Reparaturmechanismen

In diesem Kapitel wird ein Selbststabilisierungsmechanismus beschrieben, der Skip Graphen repariert, wenn es zu Knoten- und Verbindungsfehlern kommt. Zunächst werden die Bedingungen beschrieben, die für einen idealen Skip Graphen gelten. Sei  $x$  ein beliebiger Knoten. Dann gelten in einem idealen Skip Graph, für jedes Level  $i$  folgende Bedingungen:

1. Wenn  $xR_i \neq \perp$ , dann ist  $xR_i > x$ .
2. Wenn  $xL_i \neq \perp$ , dann ist  $xL_i < x$ .
3. Wenn  $xL_i \neq \perp$ , dann ist  $xL_iR_i = x$ .
4. Wenn  $xR_i \neq \perp$ , dann ist  $xR_iL_i = x$ .
5. Wenn  $i > 0$ ,  $m(x) \upharpoonright i = m(xR_{i-1}^l) \upharpoonright i$  und  $\nexists k, k < l$ ,  $m(x) \upharpoonright i = m(xR_{i-1}^k) \upharpoonright i$ , dann ist  $xR_i = xR_{i-1}^l$ .
6. Wenn  $i > 0$ ,  $m(x) \upharpoonright i = m(xL_{i-1}^l) \upharpoonright i$  und  $\nexists k, k < l$ ,  $m(x) \upharpoonright i = m(xL_{i-1}^k) \upharpoonright i$ , dann ist  $xL_i = xL_{i-1}^l$ .

**Theorem 4.1** *Jede verknüpfte Komponente der Datenstruktur ist ein Skip Graph, wenn und nur wenn alle Bedingungen 1 - 6 erfüllt sind. ([AS03])*

### 4.1 Die Invariante aufrecht erhalten

Es wird  $\perp L_i = \perp R_i = \perp$  definiert. Die Bedingungen 1 - 4 werden als invariant für Skip Graphen definiert, wenn sie sich in einem Status befinden, in dem es keine Nachrichten gibt, die noch ausgeliefert werden müssen. Sogar dann, wenn es Fehler gibt, gelten diese Bedingungen. Die Bedingungen 5 und 6 können ungültig sein, wenn es Fehler gibt. Diese Fehler können aber durch die Reparaturmechanismen korrigiert werden. Im Folgenden werden die Bedingungen 5 und 6 die  $L$  und  $R$  Nachfolgebbedingungen genannt.

**Theorem 4.2** *Gibt es keine Nachrichten, die noch abgeliefert werden müssen, dann gilt die Invariante bei Skip Graphen für das Einfügen, das Löschen und bei Knoten Fehlern. ([AS03])*

## 4.2 Die Skip Graph Bedingungen wieder herstellen

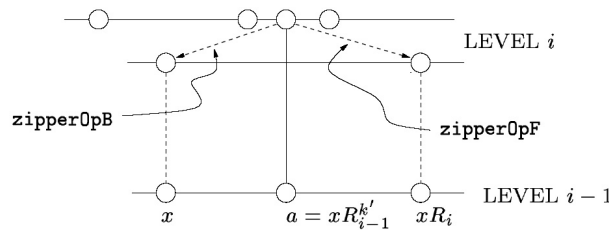
Die Nachfolgebbedingungen werden sowohl während den insert und delete Operationen verletzt, als auch wenn eine Verbindung oder ein Knoten ausfällt. Obwohl die Skip Graph Bedingungen während einer insert oder delete Operation verletzt werden könnten, sind die Nachfolgebbedingungen erfüllt, sobald keine Nachrichten anstehen oder vorbereitet werden und keine weiteren insert oder delete Operationen ausgeführt werden müssen und keine Fehler auftreten. Daher kann man sehen, dass der Reparaturmechanismus, um die Nachfolgebbedingungen wieder herzustellen nur dann benötigt wird, wenn es Knoten oder Verbindungsfehler gibt. Es wird der mögliche Fall angenommen, dass die Nachfolgebbedingungen verletzt werden können und es wird ein Reparaturmechanismus bereitgestellt, der auf beide dieser Fälle eingeht. Im Folgenden wird der Reparaturmechanismus für die  $R$  Verbindungen angenommen. Der Reparaturmechanismus für  $L$  Verbindungen ist symmetrisch. Es kann möglich sein, beide Mechanismen zu kombinieren und dadurch die Performanz zu steigern, aber in diesem Fall werden sie getrennt, um die Sache zu vereinfachen.

Es gibt zwei Fälle, in denen die  $R$  Nachfolgebbedingung verletzt ist:

1.  $xR_i = xR_{i-1}^k$ , aber  $\exists a = xR_{i-1}^{k'}$ ,  $k' < k$ ,  $m(x) \upharpoonright i = m(a) \upharpoonright i$ . Dieser Fall tritt auf, wenn zwei Knoten miteinander in den Leveln  $i - 1$  und  $i$  verbunden sind und ein neuer Knoten zwischen ihnen im Level  $i - 1$  eingefügt wird, aber auf das Einfügen in Level  $i$  vorbereitet wird. Wenn der linke Nachbar des neuen Knoten seine  $R$  Nachfolgebbedingung auf dem Level  $i$  überprüft bevor das Einfügen des neuen Knoten auf Level  $i$  komplett ist, wird eine Diskrepanz entdeckt.
2.  $xR_i \neq xR_{i-1}^k$ , für ein beliebiges  $k$ . Dieser Fall tritt in einem idealen Skip Graph auf, wenn es zu Knoten- oder Verbindungsfehlern kommt.

Jeder Fall wird im Folgenden näher betrachtet und ein Reparaturmechanismus für jede Verletzung vorgestellt.

**Fall 1:**  $xR_i = xR_{i-1}^k$ , aber  $\exists a = xR_{i-1}^{k'}$ ,  $k' < k$ ,  $m(x) \upharpoonright i = m(a) \upharpoonright i$ .



**Abbildung 4.1:** Reparaturmechanismus für Fall 1 (©[AS03])

Knoten  $a$  sollte in das Level  $i$  eingefügt werden. Dies geschieht durch das Senden folgender Nachrichten.

- Sende  $\langle zipperOpF, xR_i, i \rangle$  zu  $a$ .
- Sende  $\langle zipperOpB, x, i \rangle$  zu  $a$ .

Auf den Mechanismus zipperOpB wird in dem Algorithmus 4.1 eingegangen, der Mechanismus zipperOpF ist hierzu symmetrisch.

**Fall 2:**  $xR_i \neq xR_{i-1}^k$ , für ein beliebiges  $k$ . Abhängig davon welche anderen Knoten auf Level  $i - 1$  zu finden sind, gibt es drei Wege um diese Verletzung zu reparieren.

*Fall 2a:*  $\exists a = xR_{i-1}^k > xR_i$  und  $\nexists b = xR_{i-1}^+ < a$ , so dass gilt:  $m(b) \upharpoonright i = m(x) \upharpoonright i$ .

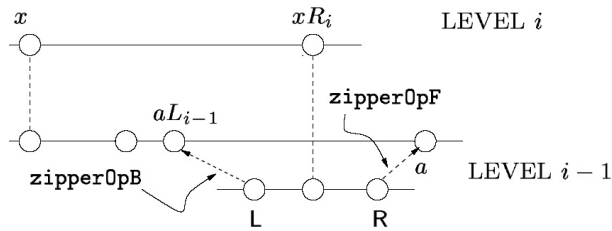


Abbildung 4.2: Reparaturmechanismus für Fall 2a (©[AS03])

Die Knoten, die mit  $a$  und  $xR_i$  auf dem Level  $i - 1$  verbunden sind, müssen in einen Ring zusammengeführt werden, indem folgende Nachrichten gesendet werden:

- Erkunde Level  $i - 1$  um das größte  $xR_i R_{i-1}^{k'} = R < a$  zu finden.
- Sende  $\langle zipperOpF, a, i - 1 \rangle$  zu R.
- Erkunde Level  $i - 1$  um das kleinste  $xR_i L_{i-1}^{k''} = L > aL_{i-1}$  zu finden.
- Sende  $\langle zipperOpB, aL_{i-1}, i - 1 \rangle$  zu L.

*Fall 2b:*  $\exists a = xR_{i-1}^k < xR_i$ ,  $m(a) \upharpoonright i = m(x) \upharpoonright i$  und  $xR_{i-1}^+ \neq xR_i$ .

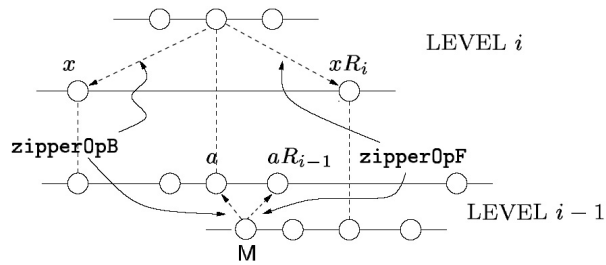


Abbildung 4.3: Reparaturmechanismus für Fall 2b (©[AS03])

Die Knoten, die mit  $a$  und  $xR_i$  verbunden sind, müssen im Level  $i$  und  $i - 1$  jeweils verbunden werden, indem folgende Nachrichten gesendet werden:

- Erkunde Level  $i - 1$  um das kleinste  $xR_i L_{i-1}^{k'} = M > a$  zu finden.
- Sende  $\langle zipperOpB, a, i - 1 \rangle$  zu M.

- Sende  $\langle zipperOpF, aR_{i-1}, i-1 \rangle$  zu M.
- Sende  $\langle zipperOpB, x, i \rangle$  zu a.
- Sende  $\langle zipperOpF, xR_i, i \rangle$  zu a.

Fall 2c:  $\exists a < xR_i, aR_i = \perp$ .

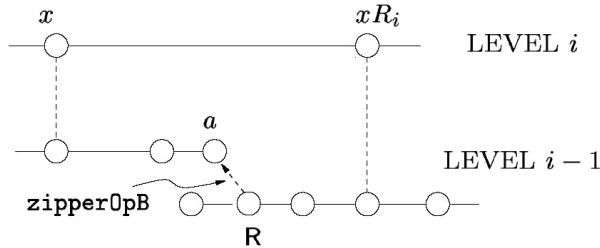


Abbildung 4.4: Reparaturmechanismus für Fall 2c (©[AS03])

Die Knoten, die mit  $a$  und  $xR_i$  auf dem Level  $i-1$  verbunden sind, müssen mittels folgender Nachrichten verbunden werden:

- Erkunde Level  $i-1$  um das kleinste  $xR_iL_{i-1}^k = R > a$  zu finden.
- Sende  $\langle zipperOpB, a, i-1 \rangle$  zu R.

**Theorem 4.3** Gibt es keine neuen Fehler mehr, so wird der beschriebene Reparaturmechanismus im Idealfall dafür sorgen, die verletzten Bedingungen wiederherzustellen, ohne dass die existierenden Verbindungen verloren gehen. ([AS03])

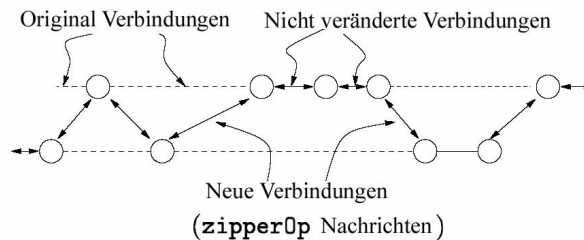


Abbildung 4.5: zipperOp Operation, um Knoten auf dem gleichen Level zu verknüpfen (©[AS03])

**Algorithmus 4.1:** zipperOpB für Knoten n

```

Beim Empfang von  $\langle zipperOpB, x, l \rangle$ :
if  $nL_l > x.key$  then
  sende  $\langle zipperOpB, x, l \rangle$  zu  $nL_l$ 
else
  tmp =  $nL_l$ 
   $nL_l = x$ 
   $nR_l = n$ 
  if tmp  $\neq \perp$  then
    sende  $\langle zipperOpB, tmp, l \rangle$  zu  $x$ 
    
```

## Kapitel 5

# Fehlertoleranz

### 5.1 Ungünstige Fehler

Die wichtigste Frage die sich bezüglich der Fehlertoleranz stellt ist, wie mehrere Knoten von der Hauptkomponente, durch den Fehler anderer Knoten, getrennt werden können, denn dieses bestimmt die Größe des Skip Graphen, nachdem der Reparaturmechanismus beendet ist.

Gegeben sei eine Teilmenge  $A$  der Knoten eines Skip Graphen. Dann wird  $\delta A$  als die Menge aller Knoten definiert, die nicht in  $A$  liegen, sondern die an  $A$  angrenzen. Des Weiteren sei  $\delta_h A$  als die Menge der Knoten definiert, die nicht in  $A$  sind, aber die mit einem Knoten in  $A$ , mittels einer Kante im Level  $h$ , verbunden sind. Es ist klar, dass gilt:  $\delta A = \bigcup_h \delta_h A$  und dass  $|\delta A| \geq \max_h |\delta_h A|$ . Der Ausdehnungsgrad einer Menge  $A$  ist  $|\delta A| / |A|$ . Der Ausdehnungsgrad eines Graphen ist der minimale Ausdehnungsgrad jeder Menge  $A$  für die gilt:  $1 \leq |A| \leq n/2$ . Der Ausdehnungsgrad bestimmt die Belastbarkeit eines Skip Graphen durch ungünstige Fehler, da eine Abtrennung einer Menge  $A$  von der Hauptkomponente voraussetzt, dass alle Knoten in  $\delta A$  versagen.

Die in [AS03] vorgestellte Strategie, die untere Grenze des Ausdehnungsgrades zu zeigen, stützt sich auf die Annahme, dass es für alle Mengen  $A$  entweder eine große Menge  $\delta_0 A$  (d.h. viele Nachbarn auf dem unteren Level des Skip Graphen) oder eine große Menge  $\delta_h A$ , für ein bestimmtes  $h$ , gewählt auf Grund der Größe von  $A$ , gibt.

Aus dieser Annahme wird in [AS03] gezeigt, dass Skip Graphen mit hoher Wahrscheinlichkeit einen Ausdehnungsgrad von  $\Omega(1 / \log n)$  besitzen. Dies bedeutet, dass sogar worst-case Fehler der Skip Graphen Struktur nur begrenzten Schaden zufügen können.

### 5.2 Zufällige Fehler

Bei zufälligen Fehlern erscheint die Situation sogar noch vielversprechender als bei ungünstigen Fehlern, wie die Experimente von [AS03] zeigen. Demnach bleiben alle Knoten in der Hauptkomponente, sogar wenn die Wahrscheinlichkeit für Knotenfehler 0.6 übersteigt. Es wird angenommen, dass viele der verlorenen Knoten zu diesem Zeitpunkt nur isoliert werden, weil die meisten ihrer Nachbarn sterben.

Bei der Suche ergibt sich aus der Tatsache, dass durchschnittlich nur  $O(\log n)$  Knoten bei einer Suche benutzt werden, dass die meisten Suchen erfolgreich sind, solange ein Anteil der fehlgeschlagenen Knoten im Wesentlichen kleiner als  $O(\log n)$  ist. Indem Fehler lokal gefunden und darüber hinaus noch redundante Kanten benutzt werden, wird die Suche sehr tolerant gegenüber kleinen zufälligen Fehlern.

## Kapitel 6

# Lastenausgleich

Zusätzlich zur Fehlertoleranz bietet ein Skip Graph eine begrenzte Form des Lastenausgleichs, indem man Krisenherde, die durch populäre Suchziele gebildet werden, beseitigt. Viele Suchanfragen, die sich an ein bestimmtes Element richten führen zu hoher Last an diesem Knoten, der dieses Element beinhaltet und an Knoten, die auf einem bestimmten Suchpfad liegen. Dennoch kann gezeigt werden, dass dieser Effekt mit der Distanz stark abnimmt. Elemente, die weit weg von einem bestimmten Ziel im unteren Teil der Liste liegen, produzieren durchschnittlich wenig zusätzliche Last.

Es werden zwei Charakteristika dieses Ergebnisses gegeben. Die Erste zeigt, dass die Wahrscheinlichkeit, dass eine bestimmte Suche einen Knoten zwischen Quelle und Ziel benutzt, umgekehrt Proportional zur Distanz des Knotens zum Ziel ist. Diese Tatsache ist wenig beruhigend wenn man an schwer beladene Knoten denkt. Da die Wahrscheinlichkeit über alle mögliche Auswahlen von Membership Vektoren gemittelt ist, kann es sein, dass sich einige bestimmte unglückliche Knoten in einem Membership Vektor befinden, durch den sie fast immer auf Suchpfade landen, die nahe zu einem sehr populären Knoten sind.

Die zweite Charakteristika bezieht sich auf dieses Problem indem gezeigt wird, dass die meisten der Lastverbreitungseffekte das Ergebnis von der Annahme eines zufälligen Membership Vektors für die Quelle der Suche sind.

**Theorem 6.1** *Sei  $S$  ein Skip Graph mit dem Alphabet  $\{0,1\}$ . Es wird eine Suche von  $s$  nach  $t$  in  $S$  angenommen. Sei  $u$  der Knoten für den gilt:  $s < u < t$  in der Schlüsselordnung. Des Weiteren sei  $d$  der Abstand von  $u$  nach  $t$ . Definiere die Anzahl der Knoten  $v$  als  $u < v \leq t$ . Dann ist die Wahrscheinlichkeit, dass eine Suche die von  $s$  nach  $t$  geht,  $u$  durchläuft kleiner als  $\frac{2}{d+1}$ . ([AS03])*

Obiges Theorem ist ein schwacher Trost für die wenigen Knoten die trotz der geringen Wahrscheinlichkeit an jeder Suche teilhaben. Glücklicherweise passieren solche Dinge nicht oft. Wir definieren die durchschnittliche Last  $L_{tu}$ , die durch eine Suche nach  $t$  auf einem Knoten  $u$  in einem gegebenen Skip Graphen  $S$  verursacht wird, als die Wahrscheinlichkeit, dass eine  $s - t$  Suche  $u$  trifft. Dies setzt die Situation in einem festen Skip Graphen voraus, in dem ein bestimmtes Ziel  $t$  für viele Suchen benutzt wird, die  $u$  treffen könnten, aber die Quelle dieser Suchen wird zufällig von den anderen Knoten im Graphen ausgesucht.

**Theorem 6.2** *Sei  $S$  ein Skip Graph mit dem Alphabet  $[0,1]$ . Des Weiteren seien  $t$  und  $u$  feste Knoten, für die gilt:  $u < t$  und  $|\{v : u < v \leq t\}| = d$ . Dann gilt für jedes andere  $\alpha \geq 0$ ,  $Pr[L_{ut} > \alpha] \leq 2e^{-\alpha d/2}$ . ([AS03])*

## Kapitel 7

---

### Fazit

In dieser Ausarbeitung wurde eine neue Datenstruktur, die Skip Graphen, für verteilte Datenspeicher vorgestellt, die mehrere wichtige Eigenschaften besitzen. Vergleicht man die Skip Graphen mit den Systemen von CAN [RFH<sup>+</sup>01] und Chord [SMK<sup>+</sup>01], so lässt sich eine bessere Performanz beobachten. So wie die Skip Graphen wird eine Zeit- und Raumspanne von  $O(\log n)$  benötigt. Doch für das Einfügen neuer Knoten benötigen die Systeme, die auf dem „Distributed Hash Table“ (DHT) Ansatz aufbauen  $O(\log^2 n)$  Zeit. Das Skip Graph System benötigt nur  $O(\log n)$  Zeit.

Einige der DHT-Systeme sind teilweise robust gegenüber zufälligen Knotenfehlern, doch ihre Verhalten wird durch das ungünstige Löschen von Knoten extrem verschlechtert ([AS03]). Wie in dem vorliegenden Dokument gezeigt wurde, wird selbst das ungünstige Löschen von Knoten der Skip Graph Struktur nur begrenzten Schaden zufügen können.

Mit dem Reparaturmechanismus des Skip Graph Ansatzes ist es möglich, Störungen in der Datenstruktur zu beheben, ohne dass weitere Fehler auftauchen. Skip Graphen unterstützen außerdem „Range Queries“ ([AS03]). Durch diese Anfragen gelingt es Knoten vor und nach einem bestimmten Schlüssel zu finden, sowie Schlüssel in einem bestimmten Intervall.

Obwohl Skip Graphen aufgrund der theoretischen Eigenschaften und der relativen Einfachheit der Struktur leicht zu implementieren sind, sind abschließende Tests ihrer Brauchbarkeit noch nicht durchgeführt worden ([AS03]).

---

# Literaturverzeichnis

- [ADS02] ASPNES, JAMES, ZOE DIAMADI und GAURI SHAH: *Fault-tolerant routing in peer-to-peer systems*. Twenty-First ACM Symposium on Principles of Distributed Computing, Seiten 223–232, 2002. [3](#)
- [AS03] ASPNES, JAMES und GAURI SHAH: *Skip Graphs*. Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, Seiten 384–393, 2003. [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [12](#)
- [Pug90] PUGH, WILLIAM: *Skip Lists: A Probabilistic Alternative to Balanced Trees*. Communications of the ACM, 33(6):338–676, 1990. [2](#), [3](#)
- [RFH<sup>+</sup>01] RATNASAMY, SYLVIA, PAUL FRANCIS, MARK HANDLY, RICHARD KARP und SCOTT SHENKER: *A scalable content.addressable network*. Proceedings of the ACM SIGCOMM, 2001. [1](#), [12](#)
- [SMK<sup>+</sup>01] STOICA, ION, ROBERT MORRIS, DAVIS KARGER, FRANS KAASHOEK und HARI BALAKRISHNA: *Chord: A scalable peer-to-peer lookup service for internet applications*. Proceedings of the ACM SIGCOMM, 2001. [1](#), [12](#)