Peer-to-Peer Information Systeme

Prof. Gerhard Weikum

7. Dezember 2004

Analyse der Evolution von P2P Systeme

WS 2004/05

Ausarbeitung

Emil Zankov - emil@zankov.de

Inhaltsverzeichnis

1 Das ideale P2P System

Die P2P-Systeme gibt es schon seit langer Zeit in verschiedenen Anwendungsszenarien - von Mediendaten Austausch bis verteilter Speicherung von Information. Um eine theoretische Analyse eines P2P System durchzuführen, müssen wir zunächst wissen, was man unter einem gut funktionierenden P2P System versteht.
Unter einem idealen P2P System, versteht man ein System, das kontinuierlich lange Zeit funktionieren kann, das ein effizientes Suchprotokoll implementiert - und somit schnelle und zuverlässige Ergebnisse liefert - das das Verbinden und Trennen von Knoten zulässt und seine Netzabdeckung in kurzer Zeit noch mal richtig setzen kann.

Die meisten P2P Systeme bieten volle Funktionalität, wenn das Verbinden von Knoten sequentiell funktioniert. Aber wie können wir ein gleichzeitiges Verbinden von Knoten unterstützen? Wenn einige Knoten das Netz verlassen, ist die Abdeckung nicht mehr ideal. Wie können wir den idealen Zustand noch mal erreichen? Was wird passieren, wenn sich das Wegfallen von Knoten mit der Zeit akkumuliert? Wie können wir eine große Anzahl von Verbindungen gewährleisten, ohne zu lange Einloggzeiten? Eine reales P2P System erreicht fast nie seinen idealen Zustand.

2 Ziele

Unser Ziel ist, ein dynamisches P2P System zu analysieren und zu versuchen zu beweisen, dass die Eigenschaften eines idealen Systems auch unter dynamischem Verbinden und Trennen von Knoten zu bewahren sind. Nachdem sogar die Hälfte aller Knoten unerwartet das Netz verlassen oder N zusätzliche neue Knoten dazukommen, sollte ein P2P System weiterhin ein zuverlässiges Suchprotokoll bieten, alle Knoten müssen immer in Verbindung bleiben und mit einer kleinen Anzahl von Idealisierungsrunden soll das System einen stabilen Zustand erreichen.

3 Entwicklungsrichtungen

Bei der Entwicklung moderner P2P Systeme haben alle Entwickler das gleiche Ziel: ein dynamisch funktionierendes System.
Einige davon (Plaxton et al. 1997) haben Verbinden und Trennen von Knoten nur dann zugelassen, wenn es gute Beziehungen zwischen den teilnehmenden Knoten und dem Netz gab. Das macht das Model unrealistisch, weil, auch wenn alle Teilnehmer sorgfältig sich vom Netz ausloggen, könnte es immer passieren, dass ein Rechner abstürzt oder eine Internetverbindung sich unerwartet trennt, und somit könnte es zu unerwünschten Netzausfällen kommen.
Eine weitere Entwicklungsarbeit (Saia et al. 2002) zielt darauf eine große Trennungsquote zu unterstützen. Sogar wenn eine konstante Fraktion von nachfolgenden Knoten, z.B. nach einem gegnerischen (Hacker)-Angriff, unerwartet das Netz verlässt, ist das System in der Lage, einen stabilen Zustand aufrechtzuerhalten, vorausgesetzt die Anzahl der verbindenden Knoten ist größer, als die Anzahl der Knoten, die das System verlassen. Dieses Protokoll birgt die Gefahr in sich, dass bei einem nicht ständig wachsenden System, d.h. keine neuen Knoten kommen dazu, oder ihre Anzahl ist kleiner, als die von den wegfallenden, das System nicht mehr richtig funktionieren wird.
Die Arbeit von Pandurangan (2001) hat ein ähnliches Protokoll, wie das "Chord" Protokoll implementiert. Allerdings ist die Menge der Information, über die ein Knoten verfügt, um das Netz aufrechtzuerhalten, O(1) im Gegensatz zu O(logN) bei "Chord". Für die Unterstützung dieses Protokolls wird jedoch ein Zentralserver eingesetzt, und dies widerspricht dem Gesamtkonzept von P2P Systemen, die dazu gedacht sind, dezentralisiert und ohne Einsatz von teueren Zentraleinheiten zu funktionieren.

4 Half-Life

Der "Half-Life" Faktor ist ein grobes Maß für den Veränderungsgrad eines Systems. Angenommen wir haben ein N-Knoten-System in Zeit t. Unter "Doubling time" versteht man die Zeit, die benötigt wird, um N zusätzliche Knoten in das System einzuloggen. "Halving time" ist die Zeit, die benötigt wird, wenn N/2 Knoten wegfallen. "Half-life" ist das Minimum von beiden Zeiten.
Ein typisches Beispiel ist das "Poisson Model" von Verbinden und Trennen. Knoten verbinden sich mit der Rate λ und trennen sich mit der Rate μ. Für ein N-Knoten-Netzwerk in Zeit t, ist die erwartete "Doubling time" N/λ and die erwartete "Halving time" $(1/\mu)\ln 2$. Der "Half-Life" Faktor ist $\min(\ln 2/\mu, N/\lambda)$. In einem System mit fester Anzahl von Knoten sind die "Halving time", "Doubling time" und "Half-Life" konstante Faktoren.
Falls ein Knoten in Zeit t im Durchschnitt weniger als k Mitteilungen per "Half-life" bekommen hat, dann wird er vom Netz mit einer Wahrscheinlichkeit von mindestens $(1 - \frac{1}{e-1})^k \approx 0.418^k$ entfernt.

5 Das Chord Suchprotokoll

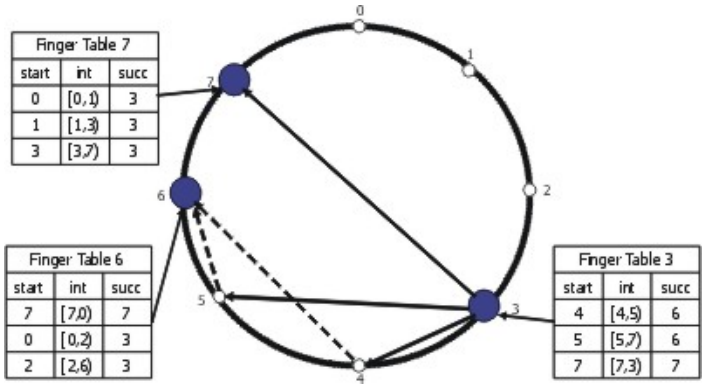

Abbildung 1: Das Chord System

Als Werkzeug für unsere Analyse werden wir ein P2P System mit "Chord" Suchprotokoll benutzen. Beim "Chord" werden die Schlüssel mit Hilfe einer Hashfunktion einem Knoten zugeordnet, der für diese Schlüssel zuständig ist. Das "Chord" System verfügt über ein ID Ring der Große 2^m. Jeder Knoten hat eine *m-bit* Nummer, die von der Hashfunktion zugewiesen wird. Jeder Knoten verfügt auch über eine Fingertabelle der Große m und Nachfolgerliste der Große *c*logN*. Mit Hilfe von diesen zwei Strukturen wird eine schnelle Suche ermöglicht. Das System toleriert auch eine große Anzahl vom Wegfallen der Knoten.

6 Probleme

Bei der Analyse müssen wir drei Hauptprobleme beachten:

Loopy States sind Probleme, die durch selten auftretende Ereignisse verursacht werden. Wir sagen, dass ein Netzwerk "schwach ideal" ist, wenn gilt `(u.successor).predecessor = u` und "stark ideal", wenn es "schwach ideal" ist und für jeden Knoten auf dem Ring gilt, dass kein weiterer Knoten v zwischen u und `u.successor` existiert (unter Beachtung der ID Nummern auf dem Ring). Wir sagen, dass ein Netzwerk "Loopy States" enthält, wenn es "schwach ideal" ist, aber nicht "stark ideal".Typische Ursachen für das Problem, sind Implementierungsfehler. Zum Beispiel: es wird über lange Zeit keine Idealisierungsprozedur auf einem Knoten angewendet. In dem Fall wird der entsprechende Knoten von dem anderen Teilnehmer für tot gehalten, obwohl er noch am Leben ist. Oder beim Verbinden und Trennen von Knoten ist es möglich, dass die Fingerzeiger falsch gesetzt werden, was zu einer inkonsistenten Suche führen kann.

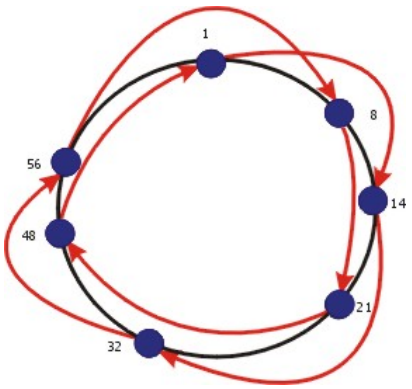

Abbildung 2: Loopy Netzwerk

Appendages treten dann auf, wenn eine große Anzahl von neuen Knoten sich gleichzeitig mit dem Netz verbinden will. Dabei stauen sie sich auf den Knoten, die schon in dem Ring sind. Die "Appenages" sind gerichtete Bäume, die auf einem Ringknoten wurzeln. Wir werden später sehen, dass dieses Problem, schnell in logarithmischer Zeit gelöst werden kann.

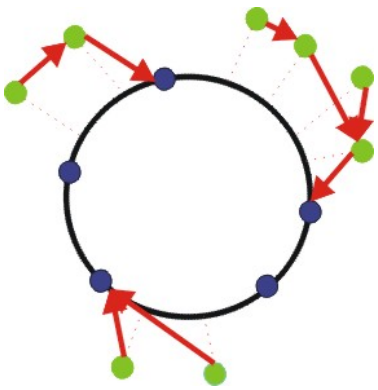

Abbildung 3: Netzwerk mit "Appendages" Knoten

Ein weiteres Problem ist das **unerwartete Trennen von Knoten**. Bei diesem Ereignis könnte es dazukommen, dass das Netz sich in zwei oder mehrere Teile spaltet, was zu einem inkonsistenten Suchprotokoll führt.

7 Analyse

7.1 Ideal State

Um unsere Analyse fortzusetzen, werden wir als erstes ein ideales "Chord" P2P System definieren. Wir sagen, dass ein "Chord" Netzwerk im Idealzustand ist, wenn:

- Jeder Knoten mit Hilfe seiner Fingertabelle und Nachfolgerliste alle anderen Knoten im Netz erreichen kann.
- Alle Koten gleichmäßig und unabhängig von einander auf dem "Chord" Ring verteilt sind.
- Alle Knoten auf dem Ring (d.h. keine "Appendages" Knoten) sind.
- Das "Chord" System keine "Loopy States" hat, sprich, dass für keinen Knoten auf dem Ring u, einen Knoten v mit $v \in (u, u.successor)$ gibt.
- Jeder Knoten verfügt über eine Nachfolgerliste der Länge $c*logN$ und diese Liste enthält genau die erste $c*logN$ echte Nachfolger von diesem beliebigen Knoten.
- Für die Korrektheit von Fingertabellen gilt, dass an der Stelle `u.finger[i]` der erste nachfolgende Knoten gespeichert ist, der Schlüssel $u + 2^{i-1}$ nachfolgt.

7.2 Failure State

Angenommen wir haben ein N-Knoten-Netzwerk und die Hälfte davon trennt sich unerwartet (unabhängig von Identifizierungsnummer und zufällig angesichts seiner Ringposition), sagen wir, dass dieses Netzwerk in "Failure State" ist.
Mit großer Wahrscheinlichkeit bleibt mindestens ein lebendiger Knoten in der Nachfolgerliste jedes Knotens. D.h. die Nachfolgerlisten sind weiterhin gültig. Für die Fingertabellen gilt auch, dass wenn `u.finger[i]` noch am Leben ist, dann ist er den ersten lebendigen Knoten, der dem Schlüssel $u + 2^{i-1}$ nachfolgt.
Nun betrachten wir folgendes Beispiel: Wir haben ein N Knoten Netzwerk in "Failure State" und keine weiteren Knotentrennungen treten auf. Unter dieser Voraussetzung ist es leicht zu zeigen, dass wir in einer kleinen Anzahl von Idealisierungsrunden, ein System in idealem Zustand bekommen können.

Beweis: Wie wir schon oben erwähnt haben, enthält jede Nachfolgerliste mindestens einen lebendigen Konten. Für einen Knoten u wird das dann das erste lebendige nachfolgende Knoten von u. Knoten u braucht also $c*logN$ mal seine Nachfolgerliste mit der von `u`.successor abzustimmen(u kopiert die Liste von seinem Nachfolger, wobei das letzte Element gelöscht wird), um die ersten $c*logN$ lebendigen Knoten einzutragen. Also nach einer konstanten Anzahl von Idealisierungsrunden hat jeder Knoten eine gültige Nachfolgerliste. Ähnlich läuft das auch bei der Fingertabelle von u. Finger `u`.finger[i] wird mit Hilfe von `u.findSuccessor(u +` 2^{i-1}`)` auf den neuesten Stand gebracht.
Diese Intuition können wir anwenden, um etwas Stärkeres zu beweisen:

Angenommen von einem N-Knoten-Netzwerk in "Failure State" trennen sich weitere N/2 Knoten, während wir O(logN) Idealisierungsrunden ausführen, so funktioniert die `findSuccessor` Prozedur zuverlässig und gibt das richtige Ergebnis in O(logN) Zeit. Das resultierende Netzwerk (nachdem die O(logN) Runden abgelaufen sind) ist wieder in "Failure State".

Beweis: Solange die Nachfolgerlisten Θ(logN) lebendige Elemente enthalten und die Knoten sich mit konstanter Wahrscheinlichkeit trennen, haben wir ein konsistentes Netzwerk.
Um die Zuverlässigkeit von `findSuccessor()` zu beweisen, betrachten wir folgendes Beispiel: Wir wenden die Prozedur `findSuccessor(k)` auf einen beliebigen Knoten u an. Der Schlüssel k befindet sich in dem Intervall des *i-ten* Fingers von u. Falls dieser Finger s noch am Leben ist, dann wird die Anfrage weitergeleitet zu dem *(i-1)-ten* Finger von s. Wenn aber der *i-ten* Finger von u nicht mehr lebt, dann müssen wir zu irgendeinem Knoten weiterleiten, dessen *i-ten* Finger noch lebt - das ist möglich, denn wir wissen, dass die Nachfolgerlisten mindestens einen lebendigen Knoten haben. Weiterhin ist es unmöglich, dass alle Nachfolger in der Liste den gleichen *i-ten* Finger haben. Mit anderen Worten, mit großer Wahrscheinlichkeit leitet Knoten u zu einem anderen Knoten, der verschiedenen *i-ten* Finger hat, weiter. Also unabhängig davon, dass u keinen *i-ten* Finger hat, besteht 1/2 Wahrscheinlichkeit, dass der *i-ten* Finger von s noch am Leben ist.
Wir springen durch eine Serie von Knoten, wo jeder Knoten verschiedene *i-ten* Finger hat. Nach maximal zwei Schritten haben wir einen Knoten mit lebendigem *i-ten* Finger erreicht. Und wir haben die Distanz verkürzt. Die Anfrage läuft so weiter, bis wir den richtigen Knoten finden. Und das funktioniert weiterhin in einem logarithmischen Faktor.

7.3 Join State

Angenommen wir haben ein N-Knoten-Netzwerk und N zusätzliche Knoten kommen dazu. Wir definieren dieses Model als "Join Model".
Hier treten die oben genannten "Appendages" Knoten auf. Da die Knoten auf dem Ring gleichmäßig und unabhängig verteilt sind, und die neu ankommenden Knoten sich zufällig an irgendeinen Ringknoten anhängen, ist mit großer Wahrscheinlichkeit die Anzahl der "Appendages" Knoten, deren ID zwischen zwei Ringknoten ist, O(logN).
Betrachten wir nun ein N-Knoten-Netzwerk mit "Appendages". Während des Ablaufs von $\Theta(log^2\text{N})$ kommen keine weiteren neuen Knoten dazu. Unter dieser Annahme wird in jeder Idealisierungsrunde ein Knoten von jedem Äppendages"Baum in den Ring integriert.

Beweis: Ein Integrierungsvorgang kann nur dann stattfinden, wenn ein "Appendages" Knoten a und ein Ringknoten v denken, dass sie denselben Nachfolger u haben. Die Idealisierungsprozedur setzt dann den Nachfolgerzeiger von v auf a und somit wird Knoten a in den Ring integriert. So in O(logN) Idealisierungsrunden werden alle "Appendages" Knoten integriert. Allerdings ist dieser Prozess keine volle Idealisierung. Um sagen zu können, dass ein Knoten voll integriert ist, sollten wir auch seine Fingertabellen richtig setzen, was uns noch weitere $O(log^2\text{N})$ Zeit kosten wird. Dann bekommen wir ein Netzwerk in idealem Zustand.

Für ein N-Knoten-Netzwerk in idealem Zustand, wo N zusätzliche Knoten sich verbinden, kann man auch feststellen, dass die Fingertabellen bezüglich die N ursprünglichen Knoten gültig sind. D.h. wir haben gültige Finger für N/2 Knoten die gleichmäßig und unabhängig verteilt sind.
Diese zwei Beobachtungen werden uns beim Beweis der folgenden Annahme sehr hilfreich sein:

Während in einem N-Knoten-Netzwerk mit "Appendages" Knoten $\Omega(log^2\text{N})$ Idealisierungsrunden ablaufen, kommen noch weitere N Knoten dazu. Unter dieser Bedingung funktioniert `findSuccessor()` Prozedur mit O(logN) Laufzeit. Das resultierende Netzwerk (nachdem die $\Omega(log^2\text{N})$ Idealisierungsrunden abgelaufen sind) ist ein Netzwerk mit "Appendages" Knoten.

Beweis: Wir unterscheiden bei so einem Prozess drei verschiedene Knotenarten: "alte Knoten", die seit mehr als $\Omega(log^2\text{N})$ Runden auf dem Ring sind. Diese sind gleichmäßig und unabhängig verteilt. "Mittelalte Knoten", die in dem Beispiel die so genannten "Appendages" Knoten repräsentieren und "neue Knoten", deren Existenz, erst während wir das Netz idealisieren, angekündigt wird. Wie vorher erwähnt, sind die Fingertabellen zuverlässig bezüglich der alten Knoten. Die neuen und mittelalten Knoten sind zufällig verteilt. Also nur O(logN) zusätzliche Knoten kommen zwischen den korrekten Knoten und den Knoten, den wir beim Suchen mit den alten Fingertabellen bekommen. Nur noch O(logN) zusätzliche Schritte werden gebraucht, um den richtigen Knoten zu finden. Wie schon erwähnt wurde, werden mittelalte Knoten in $O(log^2\text{N})$ Runden voll integriert. Danach haben wir nur alte Knoten und mittelalte Knoten Die neuen Knoten haben sich in mittelalte Knoten umgewandelt. Also sind wir wieder in einem System mit "Appendages" Knoten.

7.4 Dynamic State

Wenn die Verbindungs- und Trennungsprozesse gleichzeitig auftreten, sagen wir, dass das System sich in "Dynamic State" befindet. Dieses Model ist eine Vereinigung der beiden Modelle, die wir schon betrachtet haben. Die Bedingungen, die wir bei den obigen Betrachtungen definiert haben, gelten auch hier. Allerdings ist der konsistente Datenaustausch zwischen den benachbarten Knoten sehr wichtig (wegen des Kopierens von Nachfolgerlisten), denn andernfalls wird das Auftreten von "Loopy States" ermöglicht. Eine weitere Bedingung muss noch verstärkt werden. Die Summe von der "Appendages" Knoten muss für eine konstante Fraktion von logN aufeinander folgenden Ringknoten O(logN) sein.

Nun betrachten wir die folgende Situation:

Während in einem N-Knoten-Netzwerk mit "Appendages"- und "Failure" Knoten die D*logN Idealisierungsrunden ausgeführt werden, verbinden sich weitere N Knoten und N/2 Knoten trennen sich vom Netzwerk. Unter dieser Bedingung funktioniert `findSuccessor()` in O(logN) Zeit, und das resultierende Netzwerk ist wieder ein mit "Appendages"- und "Failure"- Knoten.

Beweis: Der Beweis läuft wie bei "Join State" und "Failure State" analog ab. Es können allerdings einige Problemsituationen auftreten, die wir noch nicht betrachtet haben. Hier werden wir wie beim "Join State" zwischen drei Knotenarten unterscheiden, und zwar alte, mittelalte und neue Knoten.

Als erstes ist es möglich, dass ein alter Knoten sich vom Netzwerk trennt. Dabei stellen wir uns die Frage, was wird mit seinem "Appendages" Baum passieren? Das Problem ist eigentlich leicht zu lösen. Von der Eigenschaft für die Summe von "Appendages" Knoten, wissen wir, dass sogar für eine Teilmenge von logN alten Knoten, die Summe der "Appendages" Knoten O(logN) ist. Also werden wir keine Eigenschaft verletzen, wenn wir die "Appendages" Knoten eines Knoten, der nicht mehr am Leben ist, mit den von dem Nachbarknoten addieren.
Zweitens wir können nicht hundertprozentig gewährleisten, dass bei einer Idealisierungsrunde immer ein mittelalter Knoten von jedem "Appendages" Baum integriert wird, solange der Vorgänger dieses Knotens wegfällt. Wir wissen aber, dass mit großer Wahrscheinlichkeit nur logN aufeinander folgenden Knoten wegfallen können. (Die Knoten sind gleichmäßig und unabhängig von einander verteilt) D.h. nach maximal logN Versuchen von einem mittelalten Knoten sich im Netz zu verbinden, wird sein Integrierungsprozess anfangen. Nachdem alle mittelalten Knoten integriert sind, brauchen wir zusätzliche log^2N Runden, um die Fingertabellen richtig zu setzen. Daraus folgt: nachdem die D*log^2N Runden abgelaufen sind, bekommen wir wieder ein System mit "Appendages" und "Failures" (die neuen Knoten sind jetzt "Appendages" Knoten).

8 Zusammenfassung

Wir haben gesehen, dass ein P2P System mit einem “Chord” Suchprotokoll auch unter dynamischen Verbindungen und Trennungen von Knoten relativ gut funktionieren kann. Die Idealisierungsschritte, die ausgeführt werden müssen, sind noch im logarithmischen Bereich, was für eine schnelle Funktionalität spricht. Weiterhin liefern die Suchoperationen das richtige Ergebnis in logarithmischer Zeit. Die Situationen, die wir hier betrachtet haben, sind sozusagen Grenzsituationen. Bei einer noch größeren Rate von Trennungen (N/2 ¡) könnte es zu einem Ausfall des Netzes kommen. Oder eine große Verbindung von Knoten (N ¡) könnte zu einem ständigen Wachsen von den “Appendages” Bäume führen und bzw. zu einer großen Wartezeit beim Einloggen ins System.

9 Literatur

- STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, "Chord: A scalable peer-to-peer lookup service for internet applications". $http://www.pdos.lcs.mit.edu/papers/chord : sigcomm01/chord_sigcomm.pdf$

- STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D., KAASHOEK, M. F., DABEK, F., AND BALAKRISHNAN, "Chord: A scalable peer-to-peer lookup service for internet applications." Tech. Rep. TR-819, MIT LCS, 2001. $http://www.pdos.lcs.mit.edu/chord/papers/$.

- DAVID LIBENNOWELL,HARI BALAKRISHNAN, DAVID KARGER, "Analysis of the Evolution of Peer-to-Peer Systems". $http://nms.lcs.mit.edu/papers/podc2002.pdf$.