ALBERT-LUDWIGS-UNIVERSITÄT FREIBURG

FAKULTÄT FÜR ANGEWANDTE WISSENSCHAFTEN

SEMINAR PEER-2-PEER NETWORKS

SYSTEM KELIPS

eingereicht am:	8 Mai 2007
von:	Mykola Potapov

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Alle Peer 2 Peer Systeme unterscheiden sich im wesentlichen in Schnelligkeit, im Speicherverbrauch (in vielen Systemen logarithmisch) und Kommunikationskosten während der Arbeit. Wir schauen uns den Versuch von Indranil Gupta, Ken Birman, Prakash Linga, Al Demers und Robbert van Renesse aus der Cornell Universität (Ithaca, USA) an, die durch das Erhöhen des Systemspeicherverbrauchs schnelle Lookups und die Systemstabilität ermöglicht haben.

1.1 Die Idee

Kelips ist basiert auf der Peer 2 Peer Distributed Hash Table. Diese Struktur erlaubt den Hosts sich ĺeise' am System an- bzw. abzumelden und Files mit bekannten Namen schnell einzufügen bzw. rauszuholen. Hier wird auch der so genannte P2P-Gossipings verwendet - das teilweise Kopieren (die Distribution) der Fileindexinformation. Dieser unterstützt eine ständige Kommunikation im Hintergrund: die Knoten quasi 'reden' mit einander. Dadurch wir es eine schnelle Konvergenz nach den Knotenwechseln so wie auch schnelle Lookups erreicht, sogar wenn die grosse Anzahl von Knoten fällt. Das System benutzt O($\sqrt{n}$) Platz am Knoten, n ist die Anzahl der Knoten im System.

2 Struktur von Kelips

Kelips besteht aus k **Affinity Groups** nummeriert von 0 bis [k-1]. Jeder Knoten liegt in einer Affinity Group an der Position, die mittels einer Hashfunktion bestimmt wird. Die Hashfunktion ordnet jedem Knotenrepräsentanten (einer IP Adresse oder Portnummer) einen Integer Wert im Bereich [0, k-1] zu. Sei n - Anzahl der Knoten im ganzen System, dann ist die Anzahl der Knoten in jeder Affinity Group etwa n/k.

Der **Node Soft State** besteht aus folgenden Komponenten:

1. **Affinity Group View** - eine Menge von anderen Knoten aus derselben Affinity Group. Die möglichen Einträge mit Informationen über die anderen Knoten sind Heartbeat Count, Round Trip Time usw.
2. **Contacts** - für jede Affinity Group im System eine kleine Menge von Knoten (konstanter Grösse) aus einer anderen Affinity Group.
3. **Filetuple** - eine Menge von Tupeln mit der Information über Namen und die Platzierung (IP Adressen von Knoten, die die Datei haben - die so genannten **Homenode**) von Files. Der Knoten speichert den Filetuple nur dann ab, wenn der homenode der Datei in derselben Affinity Group liegt. Filetuple sind auch mit einem Heartbeat-Zähler vermärkt.

Die Abbildung 2.1 zeigt die komplette Struktur des Systems. Die Einträge sind in **AVL Bäumen** gespeichert um die Effizienz der Operationen zu ermöglichen.

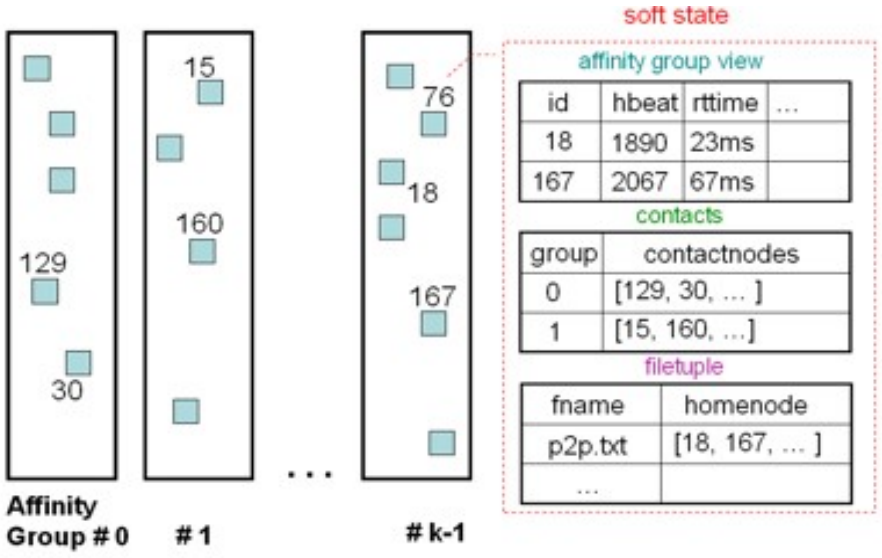

Abbildung 2.1: Struktur des Systems Kelips

3 Speicherverbrauch, Platzbegarf

Die Gesamtanzahl der Einträge am Knoten beträgt in Kelips $S(k,n) = \frac{n}{k} + c \cdot (k-1) + \frac{F}{k}$. Wobei n - Anzahl der Knoten im System, k - Anzahl der Affinity Groups, c - Anzahl der Kontakten aus der fremden Affinity Group und F - die Gesamtanzahl der Files im System ist. Für ein festes n sinkt der Wert $S(k,n)$ für $k = \sqrt{\frac{(n+F)}{c}}$. Angenohmmen ist die Gesamtanzahl der Files im System proportional zu n ist und dass c steht fest, dann bekommen wir für k einen Wert der bei O($\sqrt{n}$) liegt. Das Minimum für S(k,n) liegt auch etwa bei O($\sqrt{n}$).

4 Background Overhead

Die Einträge in Filetules, Contacst und Group Views werden periodisch innerhalb und ausserhalb der Gruppen mittels Hear tbeat Mechanism aktualisiert. Jeder Eintrag in Filetules, Contacst und Group Views ist mit einem Integer Heartbeat-Zähler assoziiert. Falls dieser während einer bestimmten Zeitperiode nicht geändert wurde, der Eintrag wird gelöscht. Das Aktualisieren vom Zähler passiert am zuständigen Knoten (für Filetuple bespielweise am Homenode). Die über änderungen verbreitet sich im ganzen System mittels gossip-style / epidemic-style protocol. Dieser Gossip-Informationsfuss erfrischt somit ständig die Einträge im ganzen System.

Gossiping innerhalb einer Affnity Group (im Falle der Aktualisierung des Heartbeat ählers): der aktuelle Stand des Zählers wird in eine Affnity Group eingefügt. Einmal erhält der Knoten die Information, fängt er an darüber eine bestimmte Anzahl von Runden (ein festes lokales Zeitinterwal) zu erzählen. Jedes Mal wählt der Knoten eine kleine konstant grosse Teilmenge der Kontaktknoten aus seiner Kontaktliste aus und leitet diese Information an die selektierten Nachbarn weiter. Somit ist die konstante Bandbreite zu erhalten. Es besteht auch eine Latenz, die sich gegen Logarithmus von der Grösse der Affnity Group variiert. Gossip-Nachrichten werden mittels Lightweigt Unreliable Protocol wie UDP verschickt. Die Auswahl der Knotenfür Gossiping geschieht nach dem motto: nehme diejenigen Knoten die am nähersten im Netzwerk sind.

über den Heartbeat-Zählerzustand soll auch innerhalb von Gruppen bekannt gegeben werden. Es wird somit während jeder Runde eine Teilmenge von Kontakten ausgewählt. Hier wird das so genannte Õzweifache GossipingÕ verwendet. Dadurch, dass das Gossiping nicht in einer sondern in mehreren Gruppen gemacht wird, steigt die Latenz um den (multiplikativen) Faktor $O(\log k)$.

Gossip-Nachrichten tragen nicht nur einen einzelnen Eintrag sondern auch äge für Filetuples bzw. Contacts und zwar inklusiv auch solche Einträge, die alt, evtl. gelöscht oder einen neuen Heartbeatt-Zähler haben. Durch das Bandbreitenlimit im System passt nicht der ganze Node Soft State in eine Gossip-Nachricht und es wird eine Aufteilung gemacht. Die maximale Aufteilung bezieht sich auf die Anzahl der Filetuples, Contacts und Group Views. Für jeden Typ des Eintrags wird die Aufteilung gleich für die neuen sowie auch für die alten Einträge gemacht. Die Einträge werden randomiziert gewählt, die nicht benutzten Partitionen werden mit der alten Information gefüllt (sobald das Refresh nicht gemacht wird). Die Grösse der Aufteilung ist unabhängig von n. Da k = $\sqrt{n}$ ist, ist der Latenzfaktor $O(\sqrt{n})$ und die Heartbeat Timeouts werden somit bis auf $O(\sqrt{n} \cdot \log^2 n)$ für Affinity View und Filetules und $O(\sqrt{n} \cdot \log^3 n)$ für Contacts Einträge variieren. Und das sind genau die Konvergenzzeiten von Nachbarnwechseln im System.

5 File Lookup und Insertion

5.1 Lookup

Der Knoten will auf eine Datei zugreifen. Er bildet den Namen der Datei in der entsprechenden Affinity Group ab (dabei wird das Hashing benutzt) und sendet eine Suchanfrage an den nah liegenden Kontakt aus seiner Affinity Group. Es wird dann weiter in den Filetuples gesucht. Das Ergebnis (die Adresse der Homenode, der die Datei hat) wird an den fragenden Knoten zurückgeliefert. Der Knoten kann jetzt auf die Datei zugreifen. Die Zeit und die Komplexität sind O(1).

5.2 Insertion

Der Knoten will eine Datei f einfügen. Er bildet den Namen der Datei in der entsprechenden Affinity Group ab und sendet eine Einfügeanfrage an den nah liegenden Kontakt aus seiner Affinity Group. Der Kontakt h holt sich randomiziert den Knoten aus seiner Affinity Group und übergibt ihm die Anfrage. Der Knoten h ist ab jetzt der Homenode. Die Datei wandelt zum Knoten h, der neue Filetuple wird erzeugt um die Datei in dem Knoten h abzubilden und wird dann in den Gossip Stream eingefügt. Zeit O(1). Komplexität O(1). Der Knoten, der das Einfügen ausgelöst hat, aktualisiert sein Filetuple um die aktuelle Information von der Datei haben.

Klar, dass die unvollständigen Filetuples und leere Contacts die Suche-/Einfügeoperationen scheitern können. Die Operationen müssen irgendwie wiederholt werden! Das wird mittels multi-hop multi-try query routing erreicht (später).

6 Protokolle und Algorithmen

Wie es in vielen Systemen organisiert ist, wird der Knoten mit dem System mit Hilfe eines Introducer-Knotens (oder einer Introducer-Gruppe) gemacht (z.B. per URL). Der neue Knoten zeigt seine Filetuples, Contacts und Vews und der Gossip Stream verbreitet schnell die Information systemweit(jeder Knoten im System schickt ab und zu einen Block mit der Information von sich und denjenigen die er kennt. Die Zeitinterwallen und die ãextra VorschriftenÒ sind im round trip time estimate in einem speziellen Gossip enthalten). Die maximale Anzahl von Kontakten ist eigentlich nicht beliebig und ist fest (falls der Gossip kein ständiges Kontakten unterstützt oder was anderes erlaubt). Es existiert auch eine **Contact Replacement Policy**. Diese beobachtet Lookups und Inserts im System. Sie achtet auch auf solche wichtige Faktoren wie Distanz zwischen den Knoten, Erreichbarkeit, Firewalls unterwegs usw.

Multi-Hop Query Routing: falls die Einfüge-/Suchanfrage verloren geht, wiederholt der Knoten die Anfrage und zwar in unterschiedlicher Weise:

1. Abfrage von verschiedenen Kontaktknoten (aus Contacts)
2. Kontaktknoten werden gefragt in ihren Affinity Groups zu schauen (während einer bestimmte TTL Zeit)
3. ein anderer Knoten aus der selben Affinity Group kann die Anfrage ausführen (falls diese sich von der File Affinity Group unterscheidet)

Query Routing führt einen randomizierten Durchlauf innerhalb der File Affinity Group in Contacts und innerhalb der Affinity Group des fragenden Knotens in (3). Die TTL-Zeit von Multi-Hop Routed Queries und der maximalen Anzahl von Versuchen (Tries) hängt von Query Lookup Success Rate (Lookup-Erfolgsbezeichner) und der Maximum Prozessing Time. Im Normalfall ist die Zeit und die Komplexität O(1). File Insertion folgt analog nach dem Multi-Try Schema. Der Unterschied liegt daran, dass die Datei dort eingefügt wird, wo die TTL Zeit abläuft. Das hilft eigentlich eine gute Lastbalance zu erzielen, was wir später an einem eispiel sehen werden. Aber in diesem Falle wächst die (normale) Insertion-Zeit auf $O(\log(\sqrt{n}))$, was aber immer noch konkurrierbar mit den heutigen Systemen ist.

7 Das Testen von Kelips

Das folgende Beispiel mit dem Testen von Kelips wurde von Entwickler an der Cornel-Universität in Ithaca, USA gemacht. Die Bilder entsprechen dem, was sie als ergebnis bekommen haben. Es wurde eine C WinAPI Implementierung von Kelips gemacht. Der Test wurde an einem Rechner mit einem emulierten Network Topology Layer gemacht (1GHz CPU, 1GB RAM, Win2K). Da die Resoursen begrenzt sind, erreicht die maximale Anzahl der Knoten im System nicht mehr als ein paar Tausend. Das Gossiping passiert jede 2 Sekunden. Jede Gossip-Nachricht beträgt 272byte. Es werden 6 Gossip-Zielknoten gewählt, 3 davon aus Kontakten.

Wir betrachten zuerst die Lastbalance. Die Abbildung 7.1 zeigt nun wie das System mit 1500 Knoten (38 affinity groups) beim Aufladen von 840, 1200 und 1900 Files aussehen wird. Sprich: wie viele Knoten haben nur eine Datei 2, 3, 8,10 usw. Files. Wie werden also alle Files im System verteilt und wie stark das System somit belastet wird. Die x-Axe beschreibt die Anzahl von Files. Die y-Axe ist die Anzahl von Knoten mit der auf der x-Axe angegebenen Anzahl von Files. Das Ergebnis ist besser als exponentiell. Die Files verteilen sich ohne Sprünge, monoton.

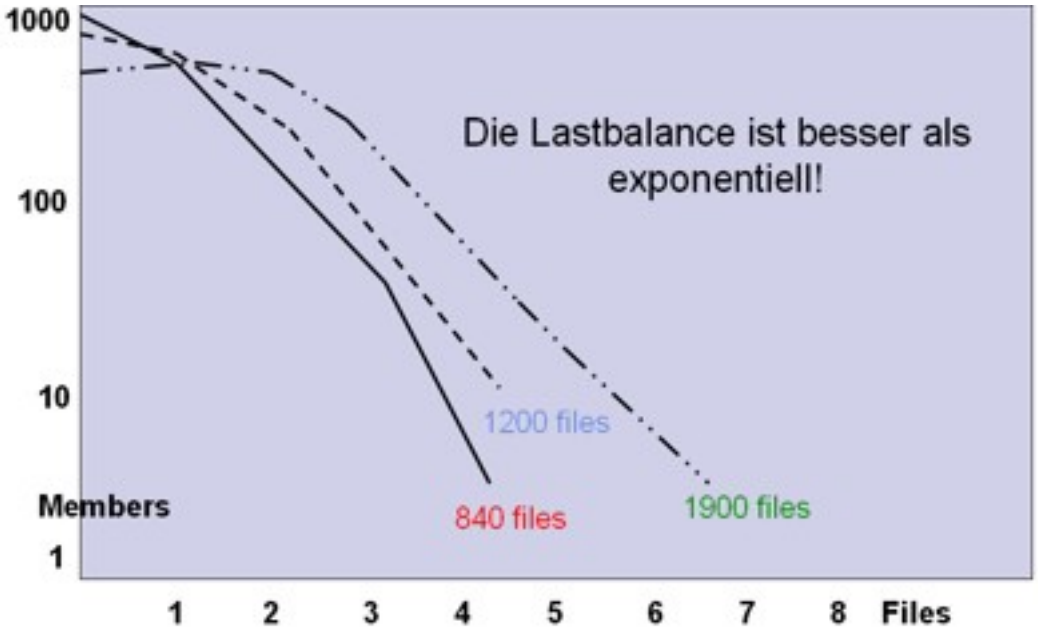

Abbildung 7.1: Lastbalance im System

File Insertion

Multi-Try Multi-Hop Schema (4 Versuche, TTL ist auf 3*log N gesetzt). 1000 unterschiedliche Files werden eingefügt. Im ersten Versuch wurden 66,2% eingefügt. 33% im zweiten. Im dritten Versuch wurden die

restlichen 0,8% ins System gebracht. Es gab keine Ausfälle oder Verluste. Nur 3 Versuche waren nötig. Bei der Insertion gab es auch also kein Problem.

Fault-tolerance

In Abbildung 7.2 wird ein System mit 1000 Knoten (30 affinity groups) betrachtet. Das lookup passiert jede zweite Sekunde. Szenario: es fallen zum Zeitpunkt 1300 500 aus 1000 Knoten aus. Was passiert dabei?

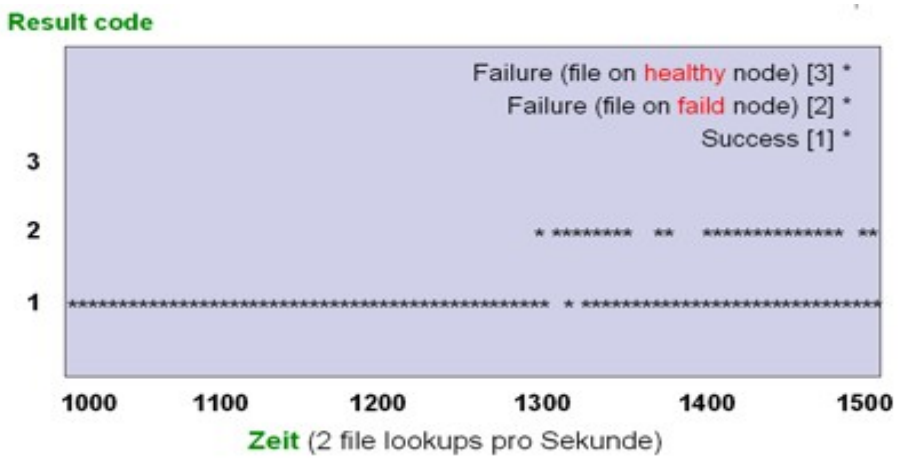

Abbildung 7.2: Fault-tolerance, Lookups

Die x-Axe beschreibt die Zeit. Die y-Axe ist die Testnummer. Der Test Nummer 1. "*"bedeutet Erfolg. Wie man sieht, ist zum Zeitpunkt 1300 kaum was passiert, das System funktioniert weiter. Der Test Nummer 2. "*"bedeutet Misserfolg, dabei fallen die Homenodes der gesuchten Datei aus. Wie man sieht, ist bis zum Zeitpunkt 1300 nichts passiert. Danach aber natürlich ist kein erfolgreiches Lookup möglich, da die Homenodes fällen. Der Test Nummer 3. "*"bedeutet Erfolg. Alle gesuchten Files waren an den "gesunden"Knoten abgespeichert, d.h. an den Knoten, die nicht ausgefallen sind. Wie man sieht, das System funktioniert überhaupt ohne Probleme.

In Abbildung 7.3 wird wieder ein System mit 1000 Knoten (30 affinity groups) betrachtet. Szenario: es fallen zum Zeitpunkt 1300 500 von 1000 Knoten aus. Was passiert dabei?
Wie man sieht, wird es zum Zeitpunkt 1380 die Stabilität erreicht und das System funktioniert ohne Probleme mit den Restlichen 500 Knoten (15 affinity groups) weiter.

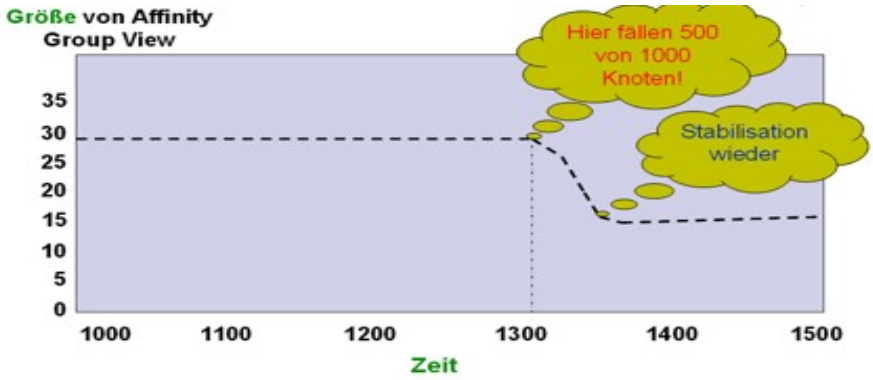

Abbildung 7.3: Fault-tolerance, Normalization

8 Zusammenfassung

Wir haben das P2P DHT basierte System Kelips kennen gelernt. Die Idee lag daran, dass der Knoten mehr Speicher verbrauchen (was eigentlich weniger als 2Mb war) und 'das richtige' Verbreitungsmechanismus (das Gossiping) benutzen soll, um schnelle Lookups und Stabilität des Systems zu bekommen, sogar wenn die grośse Anzahl der Knoten ausfällt. Der Background Overhead ist in Kelips niedrig und konstant. Die Tests der Lastbalance und Fault-tolerance beweisen die Stabilität des Systems.

Literaturverzeichnis

[1] N.T.J Bailey. *Epidemic Theory of Infectious Diseases and its Applications*. Hafner Press, 2 edition, 1975.

[2] K.P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. *Bimodal Multicast*. ACM Trans. Comp. Syst., May 1999.

[3] F. Dabek, E. Brunskill, M.F. Kaashoek, and D. Karger. *Building peer-to-peer systems with chord, a distributed lookup service*. Hot Topics in Operating Systems. Proc. 8th Wshop, May 2001.

[4] A. Demers, D.H. Greene, J. Hauser, W. Irish, and J. Larson. *Epidemic algorithms for replicated database maintenance*. Proc. 6th ACM Symp, 1987.

[5] D. Kempe, J. Kleinberg, and A. Demers. *Spatial gossip an resource location protocols*. Proc. 33rd ACM Symp, 2001.

[6] A. Rowstron and P. Druschel. *Pastry: scalable, distrubeted object location and routing for large-scale peer-to-peer systems*. Proc. IFIP/ACM Middleware, 2001.

[7] R. van Renesse, Y. Minsky, and M. Hayden. *A gossip-style failure detection service*. Proc. IFIP/ACM Middleware, 1998.