Filesharing mit Gnutella: Skalierungsprobleme eines populären P2P-Systems

Torsten Ehlers
tehlers@freenet.de

5. Juni 2002

Kurzzusammenfassung

Gnutella ist ein dezentrales P2P-File-Sharing-System mit einer wachsenden Zahl von Teilnehmern. Nach einer Kurzbeschreibung des Protokolls wird auf die Probleme Gnutellas, ausgelöst durch die Heterogenität der Teilnehmer (in Bezug auf ihre Bandbreite) sowie durch die wachsende Zahl von Nutzern, eingegangen und die Gründe hierfür erörtert. Anschließend wird ein Lösungskonzept vorgestellt, das vorgeschlagen wurde, um Gnutella besser skalierbar zu machen.

1 Einführung

1.1 Was ist Gnutella?

Bei Gnutella handelt es sich um ein verteiltes, dezentrales File-Sharing System. Im Gegensatz zu Systemen wie z.B. Napster [NAP] gibt es also bei Gnutella nicht einen oder mehrere Server, sondern jeder Teilnehmer am Gnutella-Netzwerk ist gleichwertig. Er kann also sowohl Suchanfragen von anderen annehmen, mit den eigenen geteilten Dateien vergleichen und entsprechende Antworten zurückschicken als auch selbst Suchanfragen an andere Teilnehmer versenden. Da jeder Teilnehmer somit gleichzeitig Client- als auch Serverfunktion übernimmt, verwendet man für die Teilnehmer das Kunstwort *servent*.
Ein Gnutella-Netzwerk besteht also aus kooperierenden Gnutella-Servents. Wegen der Gleichheit der Servents hat das Netz keine Hierarchie. Jeder Servent kennt nur diejenigen anderen Servents, mit denen er eine direkte Verbindung aufgebaut hat. Wie wir später sehen werden, gewährleistet dies eine gewisse Anonymität beim Durchsuchen des Netzwerks. Allerdings entsteht durch die fehlende Hierarchie eine teilweise zyklische Netzstruktur, d.h. Servents sind teilweise mehrfach auf verschiedenen Wegen miteinander verbunden, was zu unnötigem Netzverkehr führt.

1.2 Geschichte von Gnutella

Der erste Gnutella-Client trug ebenfalls den Namen Gnutella (genauer Gnutella v0.56) und wurde von Nullsoft (bekannt durch die Entwicklung von WinAmp), einer zu AOL gehörenden Firma, entwickelt. Am 14. März 2000 gab Nullsoft den Client als “open source Napster clone” zum Download frei [CLI2]. Kurze Zeit später nahm Nullsoft ihn zwar wieder vom Netz, aber

etliche andere Sites boten ihn weiterhin an. Es entstanden zahlreiche weitere Clients, die mit dem gleichen Protokoll arbeiteten und somit untereinander und auch mit dem "Ur-Gnutella" kommunizieren konnten.

1.3 Bedeutung von Gnutella

Durch die dezentrale Struktur Gnutellas hat ein Gnutella-Netzwerk keinen *single point of failure*. Allerdings sind die einzelnen Hosts in der Regel privat genutzte Computer, die häufig unvorbereitet aus dem Netz entfernt werden und Lücken hinterlassen können. Das Netzwerk unterliegt also einer ständigen Fluktuation, so waren basierend auf einer fünfmonatigen Untersuchung des Gnutella-Netzwerks [CLI2] nach 5 Stunden nur noch die Hälfte der ursprünglich gefundenen Hosts online, nach 24 Stunden nur noch 30%. Das Fehlen eines single point of failure aber ist besonders für ein Filesharing-System interessant, da beim Tausch von Audio- und Videodateien, den häufigsten Tauschobjekten in P2P-Tauschbörsen, bekanntermaßen rechtliche Probleme entstehen, die bereits zum Tod der Tauschbörse Napster geführt haben. Ein ähnliches Vorgehen gegen Gnutella ist schwierig bis unmöglich, da kein zentraler Server existiert, der abgeschaltet werden könnte. In der gleichen Untersuchung wurde die Zahl der sich täglich online befindlichen Servents im November 2000 auf etwa 10.000 - 30.000 eingeschätzt, zwei Drittel davon aus den USA, etwa 7% aus Deutschland [CLI2]. Inzwischen dürfte die Zahl der Nutzer höher liegen.

2 Das Gnutella-Protokoll

2.1 Beschreibung

Das Gnutella-Protokoll setzt auf TCP auf. Um eine Verbindung zum Gnutella-Netz herzustellen, stellt ein Host eine TCP-Verbindung zu einem ihm bekannten anderen Host her und schickt eine Connect Nachricht. Akzeptiert der Empfänger den Verbindungsaufbau so antwortet er mit einer OK-Nachricht. Damit ist eine Verbindung im Sinne des Gnutella-Protokolls aufgebaut. Von nun an erfolgt die Kommunikation über spezielle Pakete, so genannte *Deskriptoren*. Der Header eines Deskriptors besteht aus 22 Bytes und teilt sich folgendermaßen auf[CLI3] :

Feld	Bytes	Bedeutung
Descriptor ID	16	Eindeutige ID des Deskriptors
Payload Descriptor	1	Art des Deskriptors, also Ping, Pong, Query, Query Hit oder Push
TTL	1	Time To Live, Zahl der max. Weiterleitungen des Descriptors
Hops	1	Zahl der bereits erfolgten Weiterleitungen
Payload Length	3	Länge der Payload des Deskriptors in Bytes

Grundsätzlich gelten folgende Regeln beim Versenden von Deskriptoren:

1. Bei jedem Weitersenden eines Deskriptors wird das Hop-Feld um 1 erhöht
2. Entspricht der Wert von Hops dem von TTL, so wird der Deskriptor verworfen. Der Standardwert für die TTL ist 7.

3. Erhält ein Servent einen Deskriptor mit identischer ID und identischem Payload Descriptor, so wird dieser verworfen, weil er bereits einmal versandt wurde (kann auftreten, wenn Servents auf mehreren Pfaden miteinander verbunden sind).

Ein **Ping** Desktriptor wird von einem Servent versandt, um das Netzwerk nach weiteren Servents zu durchsuchen. Nachdem eine neue Verbindung aufgebaut wurde, wird normalerweise ein Ping-Deskriptor über diese Verbindung geschickt. Der Empfänger leitet diesen als Broadcast über alle offenen Verbindungen, außer diejenige über die das Ping einging, weiter. Außerdem antwortet er mit einem Pong. Ein Ping-Deskriptor besteht nur aus dem Header, die Payload ist also leer.

Pong Deskriptoren stellen die Antwort auf ein Ping dar. Ein Pong wird nicht per Broadcast versandt, sondern zum Absender des Ping geroutet. Dazu sendet der Absender und jede Zwischenstation den Pong-Deskriptor über die Verbindung zurück, über die das Ping empfangen wurde. Hierfür (und auch für die folgenden gerouteten Deskriptoren) führt jeder Servent eine Tabelle, über die er anhand der Deskriptor ID die zugehörige IP, von der er diesen Deskriptor erhielt, auffinden kann. Ein Pong erhält die gleiche ID wie das zugehörige Ping. So wird es auf genau dem Wege zurückgeroutet, auf dem das Ping durch das Netz wanderte. Ein Pong enthält im Body IP-Adresse und Port des antwortenden Clients sowie die Zahl der angebotenen Dateien und deren Gesamtgröße in KB. Der Empfänger des Pong kann nun, sofern er die Zahl seiner maximalen Verbindungen (Standardwert ist 4) noch nicht ausgeschöpft hat, eine weitere Verbindung zum Absender des Pong öffnen.

Will ein Servent eine Suche im Netz starten, so verschickt er einen **Query** Deskriptor. Neben einer Mindestgeschwindigkeit, die antwortende Servents erfüllen sollen, enthält dieser Deskriptor im Body natürlich einen Suchstring. Ein Servent, der eine Query erhält, sendet diese per Broadcast analog zur Ping-Nachricht weiter, unabhängig davon, ob er auf die Query antwortet oder nicht.

Wenn ein Servent eine Suchanfrage erhält, auf die er eine passende Datei vorrätig hat, so antwortet er mit einem **Query-Hit**-Deskriptor. Der Query-Hit wird analog zum Pong zurückgeroutet, d.h. immer nur an den direkten Vorgänger, von dem die zugehörige Suchanfrage weitergeleitet wurde. Dadurch weiß kein Servent, von wem die Suchanfrage tatsächlich kam. Es ist also auch bei Kontrolle über zahlreiche Servents nicht möglich, festzustellen von wem welche Suchanfragen gestellt werden(Anonymität bei der Suche). Query Hits enthalten im Body neben Port, IP-Adresse und Geschwindigkeit des antwortenden Servents eine Ergebnis-Menge der passenden Dateien.

Der letzte Deskriptor-Typ ist schließlich ein **Push**. Pushs werden benötigt, um Verbindungen zu Hosts hinter Firewalls bzw. mit nicht-öffentlichen IPs aufbauen zu können. Darauf soll hier nicht weiter eingegangen werden, näheres in [CLI3].

Über das Gnutella-Netz werden ausschließlich die genannten Deskriptoren verbreitet. Will ein Servent nach Erhalt eines Query-Hits eine Datei herunterladen, so baut er eine direkte HTTP-Verbindung zum entsprechenden Servent auf und teilt diesem mit, welche Datei er erhalten möchte. Der andere Servent beginnt daraufhin mit der Übertragung der Datei über diese Verbindung. Beim Herunterladen von Dateien gibt der Anfrager also seine Anonymität gegenüber dem Anbieter auf.

3 Skalierungsprobleme von Gnutella

3.1 Problembeschreibung

Wie wir gesehen haben, gibt es im Gnutella-Netzwerk keine dedizierten Server, sondern jeder Teilnehmer am Netzwerk übernimmt auch Serverfunktionen. Im Gegensatz zu bestimmten anderen P2P-Systemen wie z.B. Freenet ist die Verteilung der Dateien im Gnutella-Netzwerk ferner vollkommen unstrukturiert, d.h. es gibt keinerlei Informationen darüber, an welcher Stelle man am besten nach einer bestimmten Datei suchen soll. Der Vorteil dieser Unstrukturiertheit liegt an der leichten Anpassbarkeit an das einer ständigen Fluktuation unterworfene Gnutella-Netz. Nachteilhaft ist allerdings, daß bei der Suche kein bestimmtes Schema verfolgt werden kann, stattdessen wird eine Suchanfrage nach dem Schneeballprinzip von jedem Empfänger der Anfrage immer weiter verteilt, bis die maximale Zahl an Hops erreicht wurde. Da das Netzwerk ungeordnet und teilweise zyklisch ist, entsteht dabei natürlich auch viel unnötiger Netzverkehr, weil einzelne Anfragen denselben Host auf unterschiedlichen Wegen erreichen können. Durch dieses Broadcast-System findet Gnutella immer den kürzesten Pfad zu den Sertents, die die gesuchte Datei vorrätig haben (Verkürzung der Antwortzeit).
Allerdings wird gleichzeitig sehr viel Bandbreite im Netz verbraucht, da sich die Zahl der Query-Deskriptoren mit jedem Hop vervielfacht. Lv et al. beschreiben in [LV] gerade die Unstrukturiertheit des Netzes als das zentrale Skalierungsproblem und verweisen auf gute Skalierbarkeit bei anderen dezentralen, dafür aber strukturierten P2P-Systemen. Insgesamt führen steigende Nutzerzahlen auch zu mehr Suchanfragen (und mehr Pings) und somit zu steigendem Traffic, den jeder einzelne Host zu bewältigen hat. Es ist klar, daß es nur eine Frage der Zeit ist, bis die ersten Hosts (die mit der niedrigsten Bandbreite, also diejenigen, die über ein Modem am Netzwerk teilnehmen) den steigenden Traffic nicht mehr bewältigen können. Ein einfaches Beispiel aus [CLI1] soll das Problem verdeutlichen. Die *Network Average Query Rate* (NAQR) bezeichne die durchschnittliche Anzahl von Queries, die ein Host verarbeiten muß, also sowohl eigene als auch weitergeleitete Queries. Die folgende Abbildung zeigt im Sommer 2000 im Gnutella-Netz gemessene Average Query Rates.

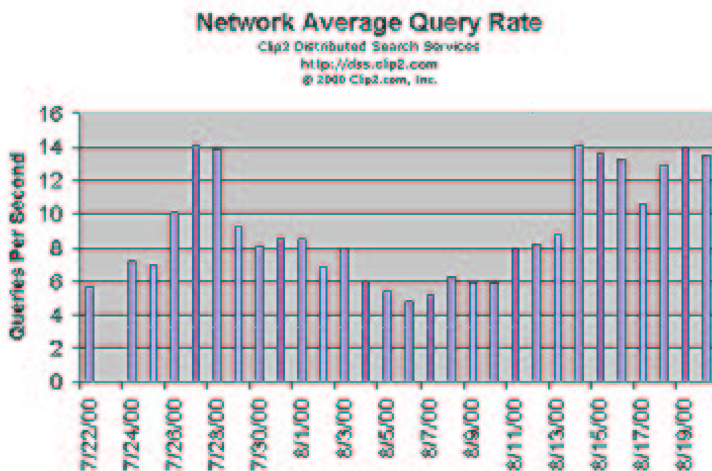

Abbildung 1: Durchschnittliche Anzahl der Queries pro Host, [CLI1]

Die Zahl der durchschnittlichen Verbindungen pro Host beträgt 3, die durchschnittliche Größe

einer Query-Nachricht 30 Bytes. Die zusätzlichen Header der unteren Protokollschichten seien mit einer Größe von 40 Bytes abgeschätzt (allein durch TCP/IP-Header gegeben). Bei einer NAQR von 10 sind also 3 x 10 x 70 = 2100 Bytes/s zu übertragen. Queries machen aber nur etwa ein Viertel des gesamten Netzwerkverkehrs aus, der Rest entfällt auf Pings, Pongs, Query Hits und Push-Deskriptoren. Somit erhalten wir ein Datenaufkommen von etwa 8400 Bytes/s oder 67,2 KBit/s. Dieses Aufkommen kann von einem Modem mit 56 KBit/s nicht mehr bewältigt werden, so daß Antworten verzögert werden bzw. gar nicht mehr ankommen. Wenn ein Host den anfallenden Datenverkehr nicht mehr bewältigen kann, bleiben ihm 3 Möglichkeiten:

- Er kann die anfallenden Daten einfach **verwerfen** - dies ist ungünstig, denn die Daten können auch Query-Hit Nachrichten enthalten, für die bereits großer Netzverkehr erzeugt wurde (Broadcast der Query-Nachricht), der nun überflüssig war, wenn die Antwort einfach verworfen wird
- Er kann die entsprechende **Verbindung schließen** - dies führt zu einem ähnlichen Problem, denn ankommende Replys auf Suchanfragen können nicht mehr zurückgeroutet werden
- Er kann die anfallenden Daten in der Hoffnung **puffern**, sie zu einem späteren Zeitpunkt versenden zu können - vorausgesetzt, dass es sich nicht nur um ein kurzzeitiges Trafficburst handelt, verschlimmert dies die Situation nur noch weiter (Verlangsamung der Antwortzeit). Schlußendlich wird auch der entsprechende Puffer überlaufen.

Hält der Datenverkehr unverändert an, ist der Host nicht mehr in der Lage, am Netz teilzunehmen. Er kann also keine Daten mehr weiterleiten und kann schlimmstensfall das Netz in 2 Teile trennen, nämlich dann, wenn er die einzige Verbindung zwischen zwei Subnetzen darstellte. Es sind also auch Hosts mit Breitbandanbindung betroffen, obwohl sie selbst evtl. gar kein Trafficproblem haben. Im Ergebnis führt dieser Effekt also zu höheren Antwortzeiten und zu weniger Treffern bei Suchanfragen für alle Netzteilnehmer.
Verschärft wird dieses Problem noch durch überflüssigen, automatisch erzeugten Netzverkehr. Es werden immer wieder Suchnachrichten der Form `a.mp3, b.mp3, c.mp3, ...` beobachtet, die als Versuch, das Netz zu indizieren, gewertet werden können.
Der TTL-Eintrag im Header, der die Ausbreitung von Nachrichten einschränken soll, begrenzt dieses Problem nur bedingt. Insbesondere ist der TTL-Wert voreingestellt und eine kurzfristige dynamische Anpassung an den derzeitigen Netzverkehr ist nicht vorgesehen. Somit ist das Netz ohne besondere Vorkehrungen nur funktionsfähig, so lange die Zahl der Servents niedrig bleibt. Ist dies nicht der Fall, so verlangsamt sich die Antwortzeit dramatisch und durch die Aufteilung des Netzes in Einzelteile wird auch die Zahl der zur Verfügung stehenden Dateien drastisch eingeschränkt. Durch die erhöhte Antwortzeit kann es zudem passieren, daß die Tabelle der Servents, über die sie Antworten zurückrouten (vgl. Beschreibung Pong-Deskriptor) überläuft, bevor eine Antwort ankommt. Dies führt zu einer weiteren Reduzierung der Nutzbarkeit.

3.2 Reliable Connection Problem

Das beschriebene Skalierungsproblem wird in [OSO] als *Reliable connection problem* bezeichnet und ist von grundsätzlicher Natur. Gnutella bedient sich TCP, also eines zuverlässigen Transportprotokolls. Will ein Servent mehr Daten versenden als es seine Verbindung hergibt, so wird TCP diese Daten zunächst puffern. Die Anwendung wird hierueber zunächst nicht

informiert und kann nicht auf die Situation reagieren sondern wird weiterhin Nachrichten an TCP zum Versenden überreichen. Das Protokoll geht also davon aus, dass es sich nur um eine kurzzeitig zu große Datenmenge handelt und der Versand bald wieder möglich sein wird. Die Anwendung auf der Absenderseite wird erst informiert, nachdem die TCP-Puffer auf der Sendeseite (meist 8-64 KB) gefüllt sind. Bei einer Puffergröße von 8KB und einer Modemanbindung mit 40KBit/s, aufgeteilt auf 4 Verbindungen, bedeutet dies bereits eine Verzögerung von etwa 8 Sekunden, bis die sendende Anwendung überhaupt etwas von dem Problem erfährt. Das Problem besteht also, weil TCP die Zuverlässigkeit der Übertragung als oberstes Ziel hat, d.h. es dürfen auf keinen Fall Daten verloren gehen, auch wenn sie dafür gepuffert oder mehrfach übertragen werden müssen. Aus Sicht von Gnutella dagegen wäre es durchaus denkbar, selektiv einzelne Nachrichten (z.B. Suchanfragen) auf einzelnen Kanälen zu verwerfen, wenn dafür die Anbindungen der Servents nicht überlastet werden.
In [OSO] wird auch die alternative Verwendung von UDP als Transportprotokoll abgelehnt, da auch die Protokolle auf Schicht 2 bereits mit Puffern arbeiten, die ebenfalls den beschriebenen Effekt aufweisen. Ferner werden natürlich willkürlich Nachrichten verworfen, was zu den oben genannten Problemen führt.

4 Lösungsansätze

Die willkürliche Verteilung der Dateien im Gnutella-Netzwerk hat den Vorteil, daß keine besonderen Maßnahmen beim Ein- und Austreten einzelner Nodes zur Beibehaltung der Struktur nötig sind. Ferner ist es mangels hierzu durchgeführter Untersuchungen noch nicht klar, ob strukturierte Systeme auch bei Suchanfragen, die nicht einen exakten Dateinamen suchen sondern gröber ausgerichtet sind, also z.B. Schlüsselwort-Suche, immer noch gut skalierbar sind. Derartige Suchanfragen sind im Gnutella-Netz häufig.
Aufgrund dieser Vorzüge des unstrukturierten Designs beschreiben Lv et al. in [LV] einen Ansatz, Gnutella unter Beibehaltung dieses Designs besser skalierbar zu machen. Das wesentliche Merkmal dieses Algorithmus' ist, daß Suchanfragen nicht nach dem Schneeballprinzip verteilt sondern von jedem Node an nur einen anderen Node weitergereicht werden. Die Auswahl des Zielnodes kann dabei nach einem Gewichtungsschema erfolgen, so daß Anfragen vorwiegend an Nodes mit breitbandiger Anbindung weitergeleitet werden. Ergebnisse erster Tests dieses Algorithmus' zeigen, daß bei Suchanfragen mit deutlichen sinkenden Antwortzeiten zu rechnen ist.
Der vorgeschlagene Algorithmus ist allerdings nicht abwärtskompatibel zum bestehenden Gnutella-Protokoll. Im folgenden soll ein Algorithmus vorgestellt werden, der diese Eigenschaft aufweist.

4.1 Ziele

Der gesuchte Algorithmus zur Lösung des Skalierungsproblems sollte folgende Kriterien erfüllen:

- Die Zahl der Hosts soll beliebig ansteigen können, ohne daß es zu einem Zusammenbruch des Netzwerks oder einer übermäßig steigenden Antwortzeit kommt
- Mit niedriger Bandbreite angebundene Hosts dürfen nicht durch Nachrichten von anderen Hosts überflutet werden
- Die verfügbare Bandbreite soll möglichst fair auf die Teilnehmer verteilt werden. Insbesondere sollen DoS-Angriffe verhindert werden

- Der Algorithmus soll kompatibel mit dem bestehenden Gnutella-Protokoll sein, d.h. Servents, die den Algorithmus nicht kennen, sollen mit "neuen", den Algorithmus implementierenden Servents zusammenarbeiten können

Der beschriebene Algorithmus besteht aus 3 Blöcken, dem **Outgoing flow control block**, dem **Q-algorithm block** und dem **Fairness block**. Diese 3 Blöcke sollen im folgenden beschrieben werden.

4.2 Outgoing flow control block

Dieser Block steuert den ausgehenden Datenfluß über eine einzelne Verbindung mit dem Ziel, die Gegenseite nicht mit Nachrichten zu überfluten und somit die Antwortzeit niedrig zu halten. Dies wird nicht nur durch Flußkontrolle, sondern auch durch gezieltes Verwerfen von die Bandbreite überfordernder Nachrichten erreicht. Der Algorithmus bedient sich hier eines einfachen Bestätigungskonzepts auf Anwendungsebene, d.h. losgelöst von den Bestätigungen auf TCP-Ebene. Nach jeweils 512 versandten Bytes wird ein PING-Deskriptor mit einer TTL von 1 eingefügt. Diese wird vom Empfänger nicht weiter verbreitet (Hop Count ist dort bereits 1) sondern lediglich mit einem PONG bestätigt. Sobald das PONG den Absender erreicht weiß dieser, daß alle vorhergehend versandten Nachrichten bereits verarbeitet worden sind; er kann jetzt weitere Nachrichten an die Gegenseite schicken. Auf diese Weise wird sichergestellt, daß niemals mehr als 512 Bytes + PING + PONG an Daten zwischen 2 Servents unterwegs sind. Während der Servent auf ein bestätigendes PONG wartet, füllt er das nächste 512 Byte große Fenster mit Daten. Was passiert wenn mehr als 512 Bytes an Daten eintreffen bevor das PONG von der Gegenseite eintrifft? In diesem Fall werden die gerouteten Replys vor den Broadcast-Requests priorisiertund über das Fenster hinausgehende Nachrichte werden verworfen. Der Sinn ist, lieber gleich die Suchanfrage, die das Netz nicht mehr verarbeiten kann, zu verwerfen, als erst eine Suche zu starten und schließlich das Ergebnis zu verwerfen. Auch innerhalb der Requests wird priorisiert, Requests mit niedriger Hop Count werden bevorzugt. Zum einen, weil das Verwerfen einer Suchanfrage mit niedriger Hop Count eine deutlichere Einschränkung des Suchraums als bei Nachrichten mit hoher Hop Count bedeutet, zum anderen weil durch dieses Schema eine Art "örtliche TTL" in das Netzwerk eingeführt wird [OSO]. Im Gegensatz zur fixen TTL wird dieser Wert dynamisch durch die örtlichen Gegebenheiten im Netzwerk, d.h. den Grad der Überlastung, "automatisch" bestimmt.

4.3 Q-algorithm block

Dieser Block befindet sich logisch am "anderen Ende", d.h. auf der Empfangsseite. Er bestimmt, welcher Teil der eingehenden Requests an die *Outgoing flow control blocks* der anderen offenen Verbindungen weitergereicht werden und welche sofort verworfen werden sollen. Die Idee hinter diesem Block ist, von vorneherein nur so viele Requests über die einzelnen Verbindungen zu versenden, daß diese hinterher auch in der Lage sind, mit der ihnen zur Verfügung stehenden Bandbreite die zu erwartende Menge an Antworten verarbeiten zu können. Dem liegt wieder zu Grunde, daß es günstiger ist, Suchanfragen bereits am Anfang zu verwerfen, als dies erst mit der Antwort zu tun (s.o.). Natürlich ist die Vorhersage der Zahl der Antworten auf eine Suchanfrage sehr schwierig, denn weder die Zahl der Antworten noch deren Größe und Verzögerung bis zum Eintreffen kann schon beim Absenden der Anfrage bestimmt werden. Hinzu kommen Einflüße des ersten Blocks, der an ihn weitergereichte Requests einfach verwerfen kann. Möglich ist nur eine statistische Analyse basierend auf dem

Request-Response-Ratio der jüngeren Vergangenheit. Somit besteht bei "statistischen Ausrutschern" immer noch die Möglichkeit einer Überflutung mit Antworten, allerdings wird deren Wahrscheinlichkeit reduziert. Der Q-Algorithmus plant maximal die Hälfte der zur Verfügung stehenden Bandbreite für die Antworten ein, so daß erst bei Antworten, die den Schnitt um mehr als das Doppelte übersteigen, Probleme entstehen.

4.4 Fairness block

Der letzte verbleibende Block stellt eine Erweiterung des *Outgoing flow control block* dar. Seine Aufgabe ist es, die zur Verfügung stehende Bandbreite für den Broadcast von Suchanfragen, die von anderen Verbindungen stammen, fair auf diese anderen Verbindungen aufzuteilen. Insbesondere soll auch DoS-Angriffen durch Überschwemmen mit Suchanfragen vorgebeugt werden. Der erste Block priorisiert Anfragen lediglich nach ihrer Hop Count. Der Fairness Block sorgt zusätzlich für eine weitere Priorisierungsstufe für Nachrichten mit gleicher Hop Count und zwar basierend auf der Verbindung, von der die Anfragen kommen. Würden zunächst alle Anfragen von Verbindung 1, dann von Verbindung 2 etc. weitergeleitet würde sicher keine Fairneß erzielt, denn Verbindungen mit höheren Nummern würden u.U. nie an die Reihe kommen. Auch eine zufällige Auswahl ist nicht sinnvoll, denn ein DoS-Angreifer könnte eine große Zahl n von sinnlosen Anfragen erzeugen, so daß bei hinreichend großem n mit beliebiger Wahrscheinlichkeit nur noch diese Anfragen weitergeleitet würden. Stattdessen arbeitet der Fairness-Block mit einem Round-Robin Schema. Für jede Verbindung wird eine Warteschlange mit Requests geführt, innerhalb der Warteschlange nach Hop-Count sortiert. Steht Bandbreite für den Versand zur Verfügung, werden Requests reihum aus den Warteschlangen entnommen und versandt.

Literatur

[OSO] Osokine, S. The flow control algorithm for the distributed 'Broadcast-Route' networks with reliable transport links. http://www.grouter.net/gnutella/flowcntl.htm, 2001

[CLI1] Bandwidth barriers to Gnutella network scalability, http://www.clip2.com/dss_barrier.html, 2000

[CLI2] Gnutella: To the bandwidth barrier and beyond, http://www.clip2.com/gnutella.html, 2000

[CLI3] The Gnutella Protocol Specification v0.4,Rev. 1.2, http://www.gnutella.co.uk/library/pdf/gnutella-protocol_04.pdf

[BOR] Bordignon, F. und Tolosa, G. Gnutella: Distributed System for Information Storage and Searching Model Description, http://www.gnutella.co.uk/library/pdf/paper_final_gnutella_english.pdf

[RIP] Ripeanu, M. und Foster, I. Mapping the Gnutella Network: Macroscopic Properties of Large-Scale Peer-to-Peer Systems

[HON] Hong, T. Performance issues in decentralized Filesharing-Networks, in Peer-to-Peer: The Disruptive Potential of Collaborative Networking, O'Reilly Verlag, 2001

[NAP] http://www.napster.com

[LIM] http://www.limewire.com

[LV] Lv, Q. et al., Can Heterogeneity Make Gnutella Scalable?, International Workshop on Peer-to-Peer Systems, 2002