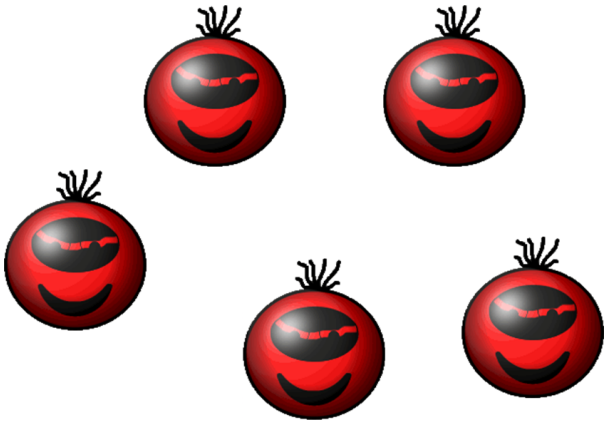
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed
Computing Group

Spamato goes P2P

Verteilen von Plug-Ins mit Peerato



Michelle Ackermann

Masterarbeit

1. April 2005 – 30. September 2005

Betreuender Professor: Prof. Dr. Roger Wattenhofer
Betreuende Assistenten: Keno Albrecht, Stefan Schmid

Vorwort

Abstract

PEERATO ist ein P2P-System zur Verteilung von Dateien an viele Downloader gleichzeitig. Zur höheren Parallelität werden die Dateien in Chunks unterteilt. Das Peerato-Protokoll erlaubt es den Peers, diese Chunks untereinander auszutauschen.

Durch Integration von PEERATO in das Plug-In System von SPAMATO wurde das Publizieren von Plug-Ins vereinfacht. Ein beliebiger Entwickler kann eigene Plug-Ins veröffentlichen. Mit dem File-Sharing System PEERATO wird dann die Zip-Datei des Plug-Ins an interessierte Benutzer weitergegeben. PEERATO kann aber auch in anderen Anwendungen eingesetzt werden, die eine Datei verteilen müssen. Vorhandene Strategien können leicht durch geeignetere ersetzt werden, je nach Anforderungen der Anwendung.

Eine wichtige Rolle bei der effizienten Verteilung von Dateien spielt die Verbindung der Peers zu einer Topologie. Strukturierte Topologien wie Baum und Hypercube wurden mit der Lower Bound bei homogenen und heterogenen Upload-Geschwindigkeiten verglichen.

Danksagung

Ich möchte mich zuerst bei meinen Eltern bedanken, die mich während meines Studiums tatkräftig unterstützt haben und immer für mich da sind.

Keno Albrecht danke ich für die Idee und Betreuung dieser Masterarbeit, den Tipps bei Problemen und den vielen spannenden Diskussionen. Die Arbeit bei ihm zu schreiben hat mir sehr viel Spass bereitet.

Ich danke Prof. Roger Wattenhofer, der es mir ermöglicht hat, meine Arbeit bei seiner Gruppe zu schreiben, und der mich zum theoretischen Teil inspiriert hat. Besonderer Dank gebührt Stefan Schmid für die Hilfe bei der Ausarbeitung dieser Überlegungen.

Bei Remo Meier möchte ich mich für die Entwicklung der SPAMATO Plug-In Architektur bedanken, die mir die Integration meiner Arbeit in SPAMATO vereinfachte.

Meinem Freund Dominik Schaffhauser danke ich für das Korrekturlesen dieser Arbeit und besonders für die Unterstützung und Geduld während meines Studiums.

Inhaltsverzeichnis

1 Einleitung

In den letzten Jahren sind verschiedene *Peer-to-Peer Systeme* (P2P-Systeme) zur Verteilung von Dateien entwickelt worden. P2P-Systeme haben verschiedene Vorteile wie *Skalierbarkeit*, *Ausfallsicherheit* und *geringe Kosten*. Im Gegensatz zu Systemen mit zentralen Servern können auch sehr grosse Dateien an viele Benutzer gleichzeitig verteilt werden.

Ein bekanntes Beispiel ist das Verteilen von Linux-Distributionen [NBRA04]. Am Tag der Herausgabe ist es fast unmöglich, die CD-Images über einen FTP-Server zu bekommen, selbst Mirror-Server sind dann kaum erreichbar. P2P-Systeme stellen hier eine Alternative dar. Sie bieten einer grossen Anzahl von Downloadern eine hohe Download-Bandbreite.

Die meisten Systeme unterteilen die Datei in mehrere kleine *Chunks*. Dies erhöht die Parallelität, da ein Peer einen fertigen Chunk bereits anderen Peers bereitstellen kann, ohne die gesamte Datei besitzen zu müssen. Zusätzlich werden die Chunks bei Erhalt mit einem Hashwert auf die Korrektheit der Daten überprüft. Dies verhindert das Weiterverteilen von falschen Daten.

Ein Nachteil des Einsatzes von P2P-Systemen ist, dass jeder Peer als potentieller Gegner angesehen werden muss, der beliebige falsche Daten senden kann. Zudem muss mit Peers gerechnet werden, die nur vom Netzwerk profitieren wollen, ohne dass sie etwas dazu beitragen. Sie könnten Chunks downloaden, ohne diese nachher an andere Peers weiterzugeben. Diese Punkte sollten beim Design eines solchen Systems berücksichtigt werden.

1.1 Motivation

Spamato ist ein kollaboratives Spam-Filter System, das in der *Distributed Computing Group* der ETH Zürich entwickelt wurde. Ausgangspunkt für die Diplomarbeit von Nicolas Burri [Bur04] war die Tatsache, dass Spam-Mails nicht nur an einen Benutzer, sondern an Millionen gesendet werden. Daher lag es nahe, dass die Benutzer zu einem Netzwerk zusammengeschlossen werden, in welchem sie die Spam-Mails kooperativ filtern.

Spamato wird ständig erweitert. Neben den kollaborativen Filtern wurden noch weitere entwickelt, wie zum Beispiel ein Bayes-basierender Filter. Alle bisher entwickelten Filter sind in [ABW05] beschrieben.

Remo Meier hat in seiner Semesterarbeit [Mei05] ein flexibles Plug-In System mit einem Update-Mechanismus entwickelt. Mittlerweile sind beinahe alle Client-seitigen Komponenten von Spamato Plug-Ins und können dynamisch ausgetauscht werden, ohne dass ein Neustart des Systems erforderlich ist. Jedes Plug-In legt einen Update-Server fest, von dem ein Update bei Bedarf per HTTP heruntergeladen werden. Zusätzlich werden Install-Server eingesetzt, um *neue* Filter und andere Plug-Ins herunterzuladen.

Gegenstand dieser Arbeit ist die Verteilung von bisher zentralen Komponenten von Spamato, um das System skalierbarer zu gestalten und eine höhere Ausfallsicherheit zu gewährleisten. Ein Bottleneck in Spamato ist der Update-Server. Bei der Herausgabe eines Updates sinkt die pro Benutzer zur Verfügung stehende Bandbreite mit der steigenden Anzahl von Spamato-Benutzern, die das Update herunterladen. Als Teil dieser Arbeit wurde daher ein effizienter Mechanismus entwickelt, um Plug-Ins zu verteilen. Da ein Plug-In in einer Zip-Datei ausgeliefert wird, kann ein File-Sharing System eingsetzt werden, um diese Datei zu verteilen. Peerato ist ein solches System und wurde in dieser Arbeit entwickelt. In Anlehnung an Tracker-basierte Systeme wie BitTorrent verteilen Benutzer das Plug-In an andere weiter, sobald sie es erhalten haben. Peerato ermöglicht es den Benutzern ausserdem, eigene Plug-Ins zu entwickeln und anderen zur Verfügung zu stellen.

1.2 Übersicht

Kapitel 2 beschreibt bestehende P2P-File-Sharing Systeme, darunter Tracker-basierte wie BitTorrent und eDonkey, und völlig dezentrale wie Kademlia, die ein DHT-Netz verwenden.

Auf die Architektur von Peerato wird in Kapitel 3 eingegangen. Zuerst werden die Anforderungen an ein verteiltes Plug-In System festgehalten. Danach wird ein Prinzip zur Verteilung erläutert, das diese Anforderungen erfüllt. Neben dem Nachrichtenprotokoll spielt in einer P2P-Anwendung auch die Verbindungsverwaltung eine wichtige Rolle. Ausserdem wird in diesem Kapitel auf eine Datenstruktur zur Unterstützung von einigen Strategien zur Auswahl von Peers und Chunks eingegangen. Schliesslich werden die Anpassungen am Plug-In System von Spamato zur Integration von Peerato gezeigt.

Gegenstand von Kapitel 4 sind einige theoretische Aspekte zur Verteilung von Chunks unter Peers. Anhand eines einfachen Modells wird eine Lower Bound entwickelt und auf heterogene Bandbreiten erweitert. Danach wird diese mit konkreten Topologien wie Baum und Hypercube verglichen.

Schliesslich werden in Kapitel 5 die Aufgaben zentraler Komponenten in Spamato angeschaut und Anforderungen an eine verteilte Variante festgelegt. Diskutiert werden für jede Komponente das Verteilen auf mehrere Server sowie das Verteilen unter Clients mit möglichen Implementierungen.

2 Bestehende File-Sharing Systeme

In den letzten Jahren wurden File-Sharing Systeme immer beliebter. Von den vielen verschiedenen P2P-Systemen besprechen wir hier nur einige. Alle hier besprochenen Systeme unterteilen grosse Dateien in mehrere kleine *Chunks*. Sie unterscheiden sich aber in der Methode, geeignete Peers kennenzulernen, und in der Entscheidung, wann welcher Chunk an welchen Peer gesendet wird.

2.1 Tracker-basierte Systeme

Ein Tracker-basiertes System stellt einen zentralen Server (*Tracker*) bereit, der den Peers hilft, sich untereinander zu verbinden. Das Prinzip des Systems besteht in der Registrierung der Peers für eine oder mehrere Dateien beim Tracker. Von diesem erhält ein Peer die Adressen von anderen Peers, die an der gleichen Datei interessiert sind. Dabei ist der Bedarf an Bandbreite beim Tracker auch bei Tausenden von Peers gering.

Trotzdem verbleibt mit dem Tracker immer noch eine zentrale Komponente im System. Wenn dieser ausfällt, kann ein neu hinzukommender Peer keine anderen mehr kennenlernen. In den vorgestellten Systemen stehen deshalb mehrere Tracker zur Verfügung, die jedoch unabhängig voneinander funktionieren und keine Daten untereinander austauschen. Abschnitt 5.1 beschreibt die Verteilung auf mehrere Tracker. Durch den Austausch von Daten kann sich ein Peer bei einem beliebigen Tracker registrieren.

2.1.1 BitTorrent

Um bei BitTorrent [Coh03] eine Datei zu verteilen, muss zuerst eine Torrent-Datei erstellt werden. Diese enthält sowohl den Namen, die Grösse und die SHA1-Hashwerte der Chunks, als auch die Adresse des Trackers, bei dem der *Seeder* mit der Datei angemeldet ist. Ein Seeder ist ein Peer, der die ganze Datei besitzt. Die Chunks haben üblicherweise eine Grösse von 256 KB.

Die Torrent-Datei wird via E-Mail oder Webseite an die Peers verteilt. Diese kontaktieren dann den Tracker und erhalten von ihm eine Liste mit zufälligen Peers, die an derselben Datei interessiert sind. Jeder dieser Peers wird dann gefragt, welche Chunks er schon hat. Hat ein Peer einen Chunk komplett und mit dem Hashwert überprüft, informiert er alle ihm bekannten Peers darüber.

Wenn ein Peer von einem anderen Peer einen Chunk anfordert, so sucht er den seltensten Chunk aus, den dieser Peer hat. Dies fördert die gleichmässige Verteilung der Chunks unter den Peers, so dass für alle Chunks etwa gleich viele Download-Quellen existieren.

Der Austausch der Chunks unter den Peers funktioniert nach dem *Tit-for-Tat* Prinzip. Jeder Peer versucht seine eigene Downloadrate zu optimieren. Dazu

füllt er vier seiner Upload-Slots mit Peers, die ihm die höchste Downloadrate liefern. Ein fünfter Upload-Slot geht an einen zufälligen anderen Peer, um herauszufinden, ob mit diesem eine noch bessere Downloadrate erzielt werden kann. Alle 30 Sekunden werden die Peers für die vier Upload-Slots aufgrund der erhaltenen Downloadraten neu berechnet und ein neuer zufälliger Peer für den fünften Upload-Slot gewählt.

Das Tit-for-Tat Prinzip erschwert das sogenannte *Freeriden*. Freerider sind Peers, die nur Chunks downloaden, aber keine uploaden. Sie können nur downloaden, wenn sie zufällig für den fünften Upload-Slot gewählt werden. Andererseits benachteiligt das Prinzip aber Analog-Modem-Benutzer, da diese eine geringe Upload-Bandbreite haben und Verbindungsabbrüche häufig sind. Ein Broadband-Benutzer wird ihn deshalb nicht in einen seiner vier Upload-Slots aufnehmen, wenn er andere Broadband-Benutzer kennt, die bessere Downloadraten liefern.

BitTorrent eignet sich gut, um populäre Dateien zu verteilen. Alle Peers konzentrieren sich auf eine bestimmte Datei und können so sehr hohe Downloadraten erreichen. Allerdings ist dies sehr Ressourcen-intensiv und skaliert nicht mit der Anzahl von Dateien. Seltene Dateien können kaum mit BitTorrent erhalten werden, da die Peers nur wenige Dateien anbieten. Weiter kann für jede Datei nur der in der Torrent-Datei eingetragene Tracker verwendet werden. Bei einem Ausfall des Trackers kann ein neu hinzukommender Peer keine anderen Peers kennenlernen.

2.1.2 eDonkey

Mit dem eDonkey Protokoll [ed2] können beliebige Dateien ausgetauscht werden. Für alle angebotenen Dateien wird ein MD5 Hashwert gebildet und zusammen mit Metadaten an einen Tracker gesendet. Andere Peers können eine Datei anhand der Metadaten finden. Die Datei wird in 9.28 MB grosse Chunks unterteilt, für die wie bei BitTorrent Hashwerte zur Überprüfung generiert werden. Diese Hashwerte werden von anderen Peers empfangen.

Eine Datei wird stets nur bei einem Tracker registriert, jedoch kann ein Peer auf allen ihm bekannten Trackern nach Dateien und anderen Peers suchen. Jeder kennengelernte Peer wird angefragt, welche Chunks er zur Verfügung hat. Er wird dann bei diesem in die Upload-Queue eingetragen und kontaktiert, sobald er der erste in der Queue ist und Daten anfordern kann.

Für das eDonkey Protokoll existieren viele unterschiedliche Implementierungen mit Verbesserungen. Mit Hilfe eines Kredit-Systems werden Uploader belohnt, indem sie in der Upload-Queue weniger lange warten müssen. Zu jedem Peer werden dazu die Anzahl der gesendeten und empfangenen Daten gespeichert und daraus ein Queue Modifier berechnet. Da sich die Peers mit einem Public-/Private-Key Handshake gegenseitig identifizieren, bleiben diese Kredite auch über mehrere Sessions erhalten. Eine weitere Verbesserung besteht darin, dass neue Peers auch durch andere Peers kennengelernt werden können. Dies wiederum entlastet die Server.

Ein Nachteil von eDonkey ist, dass die Chunks mit 9.28 MB relativ gross sind. Es dauert daher lange, bis ein neuer Peer einen kompletten Chunk hat und mit seiner Uploadrate zum Netzwerk beitragen kann. Weiter führen die

grossen Chunks möglicherweise dazu, dass mehrere Peers den gleichen Chunk vom Seeder zu downloaden beginnen, aber keiner damit fertig wird. Einige Daten werden so vom Seeder doppelt verteilt, andere gar nicht.

Im Gegensatz zu BitTorrent können mit dem eDonkey Protokoll auch seltene Dateien verteilt werden. Ein Seeder mit mehreren Dateien verspürt keinen grossen Nachteil, da er sich nur bei einem Tracker registrieren muss. Mit populären Dateien können aber höhere Downloadraten erzielt werden, da mehr Peers Chunks anbieten.

2.2 Völlig dezentrale Systeme

In Systemen ohne zentralen Server müssen einzelne Peers die Aufgabe übernehmen, andere Peers untereinander zu verbinden, die an der gleichen Datei interessiert sind. *Distributed Hash Tables* (DHTs) kommen in vielen dieser Systeme zum Einsatz und erlauben sowohl das Speichern (STORE) als auch das Finden (FIND) von Daten. Jedem Peer wird eine ID zugewiesen, welche die Position in einer bestimmten Netzwerk-Topologie festlegt. Mögliche Topologien sind zum Beispiel Ringe, Hypercubes und Bäume. Die Daten werden mit einem Key identifiziert und dem Peer mit der nächsten[1] ID zugewiesen. Um diesen Peer effizient zu finden, unterhalten die Peers Routingtabellen.

Zu beachten ist der Tradeoff zwischen der Grösse der Routingtabellen, den Kommunikationskosten und den Suchkosten. So kostet die Suche in Systemen wie Chord [SMK+01], Pastry [RD01] und Kademlia [MM02] $O(\log N)$ Hops im Netzwerk, wobei N die Anzahl der Peers ist. Bei diesen Systemen reicht es, wenn die Routingtabellen logarithmisch gross sind. In Kelips [GBL+03] sind die Suchkosten sogar konstant, dafür müssen Routingtabellen der Grösse $O(\sqrt{N})$ in Kauf genommen werden.

DHTs erlauben das Speichern von beliebigen Daten, die mit einem Key identifiziert werden können. Will sich ein Peer für eine bestimmte Datei registrieren, speichert er mit einem STORE seine Adresse beim Peer, dessen ID am nächsten beim Key der Datei liegt. Mit einem FIND erhält er Adressen von Peers, die sich für die gesuchte Datei registriert haben. Wenn jedoch viele Peers an derselben Datei interssiert sind, tritt beim für die Datei verantwortlichen Peer ein *Hotspot* auf. Er wird mit vielen STORES und FINDS überlastet. Anstelle von Adressen kann jedoch die ganze Datei oder Dateistücke im Netz gespeichert werden. Durch Caching werden Hotspots vermieden.

Durch die dezentrale Funktionsweise ist ein solches System robust gegen Ausfälle. Es tolieriert eine bestimmte Anzahl von Ausfällen von Peers, ohne dass Daten verloren gehen. Beispielsweise werden die Daten bei den k nächsten Peers gespeichert statt nur bei einem. So ist das System robust gegen $k-1$ Ausfälle. Peers, die nur kurz online sind, können aber durch das Herumschieben der Daten eine Last verursachen.

Es sind aber auch mehr mögliche Attacken zu berücksichtigen. Routing-Tabellen können gefälscht werden, oder es können viele STORES mit falschen Daten aufgerufen werden. Weiter können zusammenarbeitende Gegner-Peers

[1] Das Distanzmass zwischen ID und Key wird vom jeweiligen System definiert.

ihre IDs so wählen, dass sie für bestimmte Daten alleine verantwortlich sind, und diese dann fälschen oder „vergessen“. Dies kann aber schon dadurch verhindert werden, dass die IDs nicht frei wählbar sind, sondern zum Beispiel aus dem Hashwert der IP-Socket-Adresse[2] gebildet werden. Genauso werden für die Keys ein Hashwert der Daten bestimmt.

Ein weiteres Problem tritt auf, wenn viele Peers an derselben Datei interessiert sind. Der für die Datei verantwortliche Peer ist dann mit den vielen STORES und FINDS überlastet. Schliesslich muss das Bootstrapping-Problem gelöst werden: Ein neu hinzukommender Peer muss schon mindestens einen anderen Peer kennen.

2.2.1 Kademlia

Das Kademlia Protokoll [MM02] ist ein DHT-System und weist Peers und Daten eine 160-bit ID zu. Ein Key wird dem Peer zugewiesen, dessen ID die kleinste Distanz zum Key hat. Distanzen werden als XOR der IDs berechnet.

Jede empfangene Nachricht wird zum Aktualisieren der Routingtabellen benutzt. Die Routingtabellen bestehen aus 160 k-Buckets[3] mit Peers, von denen Nachrichten empfangen wurden. Im i-ten Bucket sind Peers mit Distanzen zwischen 2^i und $2^i + 1$ gespeichert. Die Peers in den Buckets sind nach dem Timestamp der letzten empfangenen Nachricht sortiert. Wenn eine Nachricht von einem neuen Peer empfangen wird und das Bucket voll ist, wird ein Ping an den Peer im Bucket gesendet, von dem am längsten keine Nachricht mehr empfangen wurde. Er wird nur dann durch den neuen Peer ersetzt, wenn er nicht erreicht werden kann. Dies berücksichtigt die Beobachtung, dass Peers, die schon lange im System sind, mit hoher Wahrscheinlichkeit noch länger im System bleiben und deshalb nicht ersetzt werden sollten. Das System kann so nicht durch das Einführen von vielen neuen Gegner-Peers attackiert werden.

Die wichtigste Prozedur ist das Finden der k nächsten Peers zu einer gegebenen ID. Dazu werden aus dem nächsten nicht-leeren k-Bucket α ausgesucht. Diese antworten mit einer Liste mit k ihnen bekannten Peers, welche die kleinste Distanz zur ID haben. Rekursiv werden nun die k nächsten Peers aus den Antworten ausgesucht und α davon gefragt. Die Prozedur terminiert, wenn alle k nächsten Peers angefragt wurden und es in den Listen keine näheren Peers gibt. STORES und FINDS werden direkt an diese k nächsten Peers gesendet. Durch das STORE auf k Peers wird Replikation erreicht. Damit keine Daten verloren gehen, sollte der Ausfall von beliebigen k Peers im gleichen Zeitintervall sehr unwahrscheinlich sein.

Ein neuer Peer muss mindestens einen Peer kennen. Er sucht nun nach den k nächsten Peers zu sich selbst und frischt so seine Buckets auf. Wenn ein anderer Peer Daten gespeichert hat, deren Key näher beim neuen Peer als bei ihm selbst liegen, repliziert er diese Daten beim neuen Peer.

Da keine Nachrichten geroutet, sondern immer direkt an den Adressaten gesendet werden, sind Routing-Attacken schwierig. Es können lediglich falsche

[2] IP-Adresse und Port-Nummer
[3] Datenbehälter, der maximal k Elemente beinhaltet

Adressen zurückgeschickt werden. Bei der Anfrage im nächsten Schritt sieht man aber sofort, dass diese nicht antworten.

Wenn mit Kademlia Dateistücke im Netz zu speichern sind, können Hotspots mit Caching vermieden werden. Ein Peer, der mit einem FIND ein Dateistück holt, speichert dieses gleich noch beim letzten Peer, der es nicht hatte.

In einigen Implementierungen von eDonkey und BitTorrent kann inzwischen auch mit Kademlia gesucht werden. Wenn ein Tracker ausfällt, können so trotzdem neue Peers gefunden werden.

3 Peerato Architektur

3.1 Anforderungen

Ein verteiltes Plug-In System unterstützt sowohl die Verteilung neuer Plug-Ins als auch Updates von existierenden Plug-Ins. Ein Update soll an möglichst viele Benutzer gleichzeitig gesendet werden können, insbesondere wenn die Anwendung Updates automatisch holt. Eine zentrale Lösung eignet sich daher kaum, da zu Spitzenzeiten selbst die Mirrors des Servers überlastet sind.

Es muss daher untersucht werden, in welcher Form Plug-Ins dezentral verteilt werden können. Um mögliche Lösungen gegeneinander abwägen zu können, müssen zuerst die *Anforderungen* an ein solches System festgelegt werden:

Erreichbarkeit Ein Plug-In ist erreichbar, wenn ein Peer online ist, der das Plug-In anbietet.

Skalierbarkeit Ein Plug-In muss auch dann erreichbar sein, wenn es von vielen Peers gleichzeitig angefordert wird. Eine Suche für eine Download-Möglichkeit darf das Netz nicht überlasten und muss effizient sein (höchstens logarithmisch lange in der Anzahl von Peers). Das Downloaden sollte immer direkt sein, so dass keine Daten unnötig zu einem Peer gesendet werden müssen, der diese gar nicht braucht.

Lastverteilung Ein Peer darf nicht überlastet werden und sendet stets gleichzeitig an eine begrenzte Anzahl von Peers. Jeder Peer, der vom Netz profitiert, trägt auch etwas dazu bei. Ein Peer sollte möglichst schnell zum Netz beitragen können und daher auch schon vor dem kompletten Downloaden des Plug-Ins die Möglichkeit haben, Teile des Plug-Ins an andere Peers weiterzugeben. Ebenfalls sollte ein Peer von mehreren anderen Peers gleichzeitig downloaden können.

Sicherheit Das System muss mit Gegnern rechnen, die beliebige falsche oder gar keine Daten senden.

Ausfall der Peers Das System muss mit Ausfällen von Peers fertig werden. Peers können nach dem Downloaden sofort ausfallen, aber auch schon während dem Downloaden.

Unterstützung von Firewalls Peers, die hinter einem Router oder einer Firewall sind, sollten die Möglichkeit haben, Plug-Ins zu downloaden und auch an andere weiterzugeben.

Dateigrösse Ein Plug-In ist kleiner als beispielsweise eine ISO-Datei, die mit BitTorrent verteilt wird. Das System sollte deshalb auch zum Verteilen von kleineren Dateien geeigent sein.

Seltene Downloads Wenn Downloads selten sind, aber genügend andere Peers das Plug-In haben, soll der Downloader seine gesamte Download-Bandbreite nutzen können.

Unterschiedliche Bandbreiten Besonders das Downloaden von kleinen Plug-Ins ist auch für Analog-Modem-Benutzer interessant. Deshalb sollten sie nicht durch ihre relativ geringe Upload-Bandbreite benachteiligt sein.

3.2 Prinzip

Um die Probleme eines völlig dezentralen Systems zu vermeiden (siehe dazu Abschnitt 5.1.2), wurde für Spamato eine Tracker-basierte Lösung entwickelt. Peerato besteht aus dem Tracker-Server und dem Peerato Plug-In für die Peers. Die Peers fragen den Tracker an, um andere Peers, die an der selben Datei interessiert sind, kennenzulernen. Dies braucht beim Tracker nur sehr wenig Bandbreite.

Wie bei den meisten File-Sharing Systemen wird die Datei in Chunks aufgeteilt. Die Chunks können von verschiedenen Peers erhalten werden. Die bereits erhaltenen Chunks sind für andere Peers schon verfügbar, bevor der Peer die komplette Datei hat. Somit kann eine Datei schnell weiterverteilt werden.

3.3 Nachrichtenprotokoll

Alle Nachrichten in Peerato werden im XML-Format verschickt, so dass das Protokoll auch mit einer anderen Programmiersprache implementiert werden könnte. Durch Komprimierung können die relativ grossen XML-Nachrichten verkleinert werden.

Um Dateien beim Tracker zu registrieren und andere Peers kennenzulernen, wird das folgende Protokoll (Abbildung 3.1) verwendet:

1. Der Peer sendet dem Tracker einen *PeerListRequest*. Ein PeerListRequest enthält den Port, unter dem er von anderen Peers kontaktiert werden kann und alle zu registrierenden *FileIds*[1] mit den Angaben, ob die Datei komplett ist und wieviele Peers er kennt.

2. Der Tracker registriert den Peer für die angegebenen Dateien und sendet ihm eine *PeerListAnswer*. Eine PeerListAnswer enthält eine Liste von Peer-Adressen für alle nicht kompletten Dateien, zu denen er weniger Peers kennt, als die vom Tracker festgelegte Anzahl.

[1] Registriert werden sowohl angebotene komplette Dateien, als auch Dateien, zu denen noch Chunks fehlen.

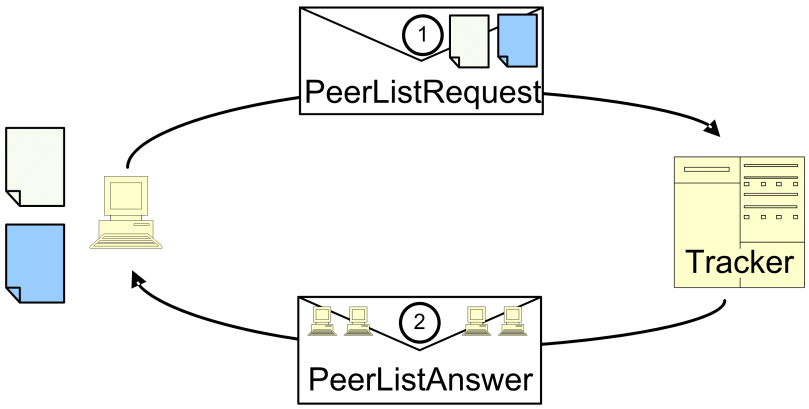


Abbildung 3.1: Nachrichtenaustausch zwischen Peer und Tracker.

Ein Peer lernt neue Peers kennen, wenn er eine PeerListAnswer vom Tracker erhält, oder ein anderer Peer eine Verbindung zu ihm aufbaut. Dann wird das folgende Protokoll (Abbildung 3.2) ausgeführt:

1. Peer A schickt dem neuen Peer B einen *ChunkListRequest*. Dieser enthält die FileIds aller nicht kompletten Dateien.

2. B antwortet mit einer *ChunkListAnswer*. Diese enthält alle FileIds, die er selbst hat, mit einem Bitfeld von Chunks, die er von der Datei schon hat.

3. Wenn A einen Chunk von B noch nicht hat, sendet er ihm einen *DownloadRequest*. B trägt ihn dann in seine Upload-Queue ein. Wenn es keine freien Upload-Slots hat und sonst keine Nachrichten gesendet werden müssen, kann die Verbindung bis auf weiteres geschlossen werden.

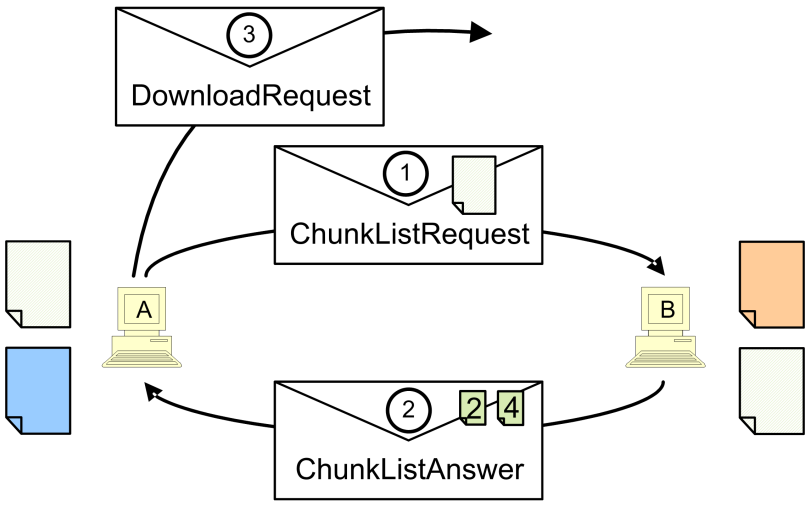


Abbildung 3.2: Nachrichtenaustausch zum Kennenlernen eines neuen Peers.

Wird bei einem Peer B ein Upload-Slot frei, so wird dieser mit einem Peer A aus der Upload-Queue gefüllt. Dann wird folgendes Protokoll (Abbildung 3.3) ausgeführt:

1. B sendet A eine *UploadOffer*.
2. Bei Erhalt einer UploadOffer sucht sich A einen Chunk aus, den B anbietet. Er fordert den Chunk mit einem *ChunkRequest* an. Wenn B keine passenden Chunks anbietet, so wird stattdessen ein *NotInterested* gesendet, worauf A von B aus dem Upload-Slot entfernt wird.
3. B sendet A den Chunk in einer *ChunkAnswer*. Nach erfolgreichem Senden der Daten wird A automatisch wieder in die Upload-Queue von B eingetragen.
4. Nach dem Empfang eines Chunks wird dieser mit dem Hashwert überprüft und abgespeichert. A informiert nun diejenigen seiner Nachbarn[2], die den Chunk noch nicht haben, über den Erhalt des Chunks. Dazu sendet er ihnen ein *HaveChunk*, mit der Angabe, welchen Chunk er gerade erhalten hat. Die Nachbarn werden dann automatisch in die Upload-Queue von A eingetragen.

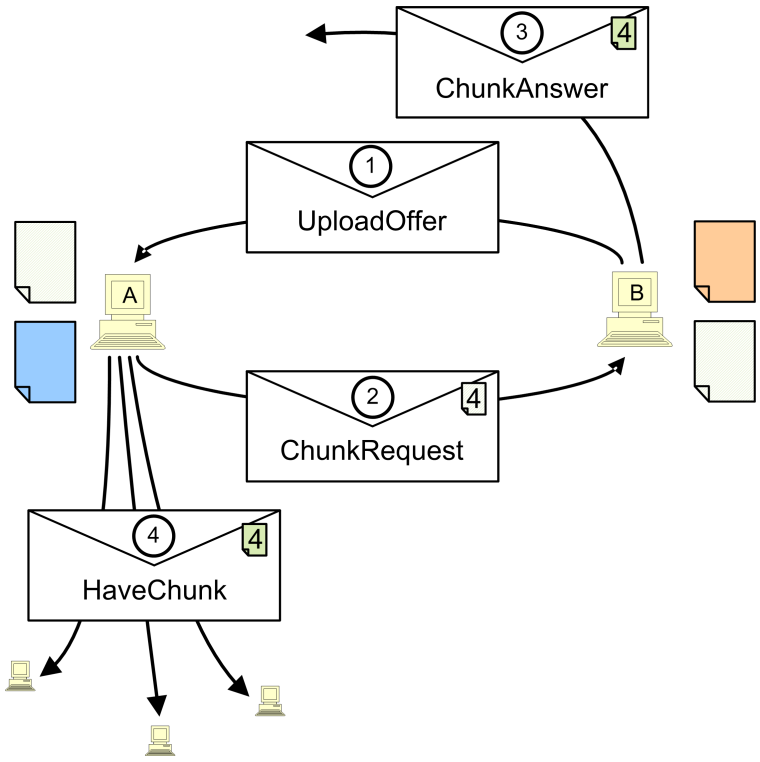


Abbildung 3.3: Nachrichtenaustausch zwischen zwei Peers zum Downloaden eines Chunks.

[2] Alle kennengelernten Peers sind Nachbarn eines Peers.

3.4 Verbindungsverwaltung

In einer P2P-Anwendung werden viele Verbindungen zu anderen Peers aufgebaut. Daher ist das Starten eines Threads für jede Verbindung nicht geeignet. Mit Java NIO wird jedoch nur ein Thread benötigt, um alle Verbindungen zu verwalten. Dadurch ist auch der Synchronisationsaufwand deutlich geringer. Die Java NIO Klassen sind in [Hit02] sehr gut beschrieben. In der Implementierung von PEERATO werden die Klassen Selector, SelectionKey, ServerSocketChannel, SocketChannel und ByteBuffer verwendet. Abbildung 3.4 gibt einen Überblick über die in den nächsten Abschnitten besprochenen Klassen *ConnectionManager* und *Connection*.

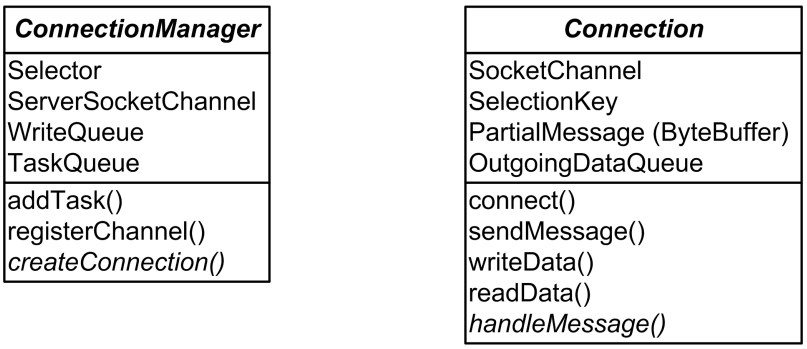


Abbildung 3.4: Klassen zur Verbindungsverwaltung in Peerato.

3.4.1 ConnectionManager

In PEERATO verwaltet die Klasse *ConnectionManager* alle Verbindungen mit einem *Selector*. Zuerst wird ein *ServerSocketChannel* erstellt und beim Selector registriert, um ankommende Verbindungen annehmen zu können. Wenn der Selector einen ankommenden *SocketChannel* meldet, erstellt der ConnectionManager daraus eine *Connection* und registriert den Channel beim Selector für Lese-Operationen mit der Connection als Attachment.

3.4.2 Connection

Die Klasse Connection implementiert Methoden, um Daten des SocketChannels zu lesen und zu schreiben. Der ConnectionManager ruft die Methode *readData* der Connection auf, wenn der SocketChannel zum Lesen bereit ist, und *writeData*, wenn der SocketChannel zum Schreiben bereit ist und für Schreib-Operationen registriert ist.

Da Lese-Operationen nicht-blockierend sind, speichert die Connection in einem *ByteBuffer* das bisher gelesene. Die komplette Nachricht[3] wird dekomprimiert und von der konkreten Connection mit *handleMessage* bearbeitet.

[3] Um das Ende einer Nachricht zu erkennen, wird zuerst die Länge der Nachricht gesendet.

Schreib-Operationen sind ebenfalls nicht-blockierend. Eine ausgehende Nachricht wird deshalb nicht sofort gesendet, sondern in die *OutgoingDataQueue* der Connection eingetragen. Danach wird der Channel für Schreib-Operationen registriert, so dass der ConnectionManager die Methode *writeData* aufruft, wenn der Channel zum Schreiben bereit ist. Mit der Implementierung des *MessageSentListener*-Interfaces kann sich ein Objekt über das erfolgreiche Versenden einer Nachricht informieren.

3.4.3 Begrenzung der Upload-Geschwindigkeit

Das Schreiben ist an eine maximale *Upload-Geschwindigkeit* gebunden. Statt dem sofortigen Aufruf von *writeData* wird eine schreibbare Connection in die *WriteQueue* beim ConnectionManager eingetragen und für Schreib-Operationen deregistriert. Dann werden jeweils 1 KB Daten der ersten Connection aus der WriteQueue geschrieben. Wenn die OutgoingDataQueue noch Daten enthält, wird die Connection wieder für Schreib-Operationen registriert. Dann wird solange mit dem nächsten Schreiben gewartet, wie mit der gegebenen Upload-Geschwindigkeit benötigt würde, um die Daten zu senden.

3.4.4 Begrenzung der Anzahl Verbindungen

Ausgehende Verbindungen benötigen die IP-Socket-Adresse des Empfängers und werden ebenfalls beim Selector registriert. Um aber nicht zu viele Verbindungen gleichzeitig zu öffnen, ist die Anzahl der Connections limitiert. Die überzähligen ausgehenden Connections werden beim ConnectionManager in die ConnectionQueue eingetragen. Nachrichten können dann bereits in die OutgoingDataQueue eingetragen werden, gesendet werden sie aber erst, wenn die Anzahl der aktiven Connections unter das Maximum gesunken ist. Dann wird *connect* aufgerufen, worauf die Verbindung aufgebaut und der Channel für Schreib-Operationen registriert wird.

3.4.5 Synchronisation

Um den Synchronisationsaufwand gering zu halten, implementiert der ConnectionManager das Interface *TaskQueue*. TaskQueue stellt die Methode *addTask* zur Verfügung, womit der Task beim ConnectionManager einer synchronisierten Queue hinzugefügt wird. Andere Threads, die zum Beispiel eine Verbindung zum Senden einer Nachricht benötigen, führen das nicht sofort aus, sondern fügen den entsprechenden Task in die TaskQueue ein. Regelmässige Tasks, wie das Senden von PeerListRequests an den Tracker oder Entfernen von alten Objekten aus den Datenstrukturen, werden von einem Timer ebenfalls in die TaskQueue eingetragen, anstatt sie sofort auszuführen. Der ConnectionManager führt alle anstehenden Tasks aus, bevor er mit der *select*-Methode des Selectors auf das nächste Ereignis[4] wartet.

[4] Ein Ereignis ist entweder ein Channel mit einer anstehenden Operation oder ein Wakeup-Aufruf eines Tasks.

3.4.6 Konkrete Implementierungen

Der *TrackerServer* erweitert den ConnectionManager, indem er die Methode *createConnection* implementiert. Sie liefert für einen ankommenden SocketChannel eine *PeerConnection*, eine konkrete Implementierung der Connection. Sie bearbeitet PeerListRequest-Nachrichten.

Der *PeerServer* erweitert ebenfalls den ConnectionManager. Ein Peer unterscheidet zwischen drei verschiedenen Connections. Mit der *TrackerConnection* wird ein PeerListRequest an den Tracker gesendet und eine PeerListAnswer empfangen und bearbeitet.

Eine *PeerConnection* stellt eine Kommandoverbindung zu einem anderen Peer dar und bearbeitet *PeerMessages*. Auf der Kommandoverbindung werden wie bei FTP keine Daten gesendet, damit Kommandos schnell gesendet werden können, auch wenn gleichzeitig viele Daten zum selben Peer unterwegs sind.

Eine *DataConnection* erweitert die Klasse PeerConnection und stellt eine Datenverbindung zu einem anderen Peer dar. Auf der Datenverbindung werden ChunkAnswer Nachrichten gesendet und empfangen. Für ankommende Datenverbindungen wird ein zusätzlicher ServerSocketChannel auf einem anderen Port geöffnet. Die Datenverbindung wird normalerweise vom Empfänger eines Chunks aufgebaut. Kommt sie aber aus irgend einem Grund nicht zustande (Firewall), baut der Sender selbst eine Datenverbindung auf und schickt darauf den Chunk. Hat der Empfänger zu einem Zeitpunkt kein Interesse mehr am Chunk, kann er den Download jederzeit durch Schliessen der Datenverbindung abbrechen, ohne dass Nachrichten auf der Kommandoverbindung verloren gehen.

3.5 Datenverwaltung

3.5.1 Tracker-Server

Der Tracker muss wissen, welche Peers sich für welche Datei registriert haben, um einem Peer bei Bedarf neue Nachbarn zuzuteilen. Dazu wird eine *NeighborSelectionStrategy* mit den folgenden Methoden implementiert:

getPeer liefert zu einer gegebenen IP-Socket-Adresse den Peer.

getNeighbors liefert zu einem gegebenen Peer mit fehlenden Dateien neue Nachbarn, die sich für mindestens eine der fehlenden Dateien registriert haben.

addPeerToFile registriert den Peer für die gegebene Datei.

RandomSelector ist eine mögliche Strategie und unterhält eine Map mit allen registrierten FileIds, die auf eine Liste von Peers verweisen. Die Liste enthält jene Peers, die sich für die Datei registriert haben. Die Strategie gibt eine zufällige Auswahl von Nachbarn zurück, die der Peer noch nicht kennt. Die Peers sind in einer Map mit der IP-Socket-Adresse als Key gespeichert. Zu den Peers sind folgende Daten vorhanden:

- IP-Socket-Adresse

- Timestamp der letzten Registration
- Nachbarn, die dem Peer schon gesendet wurden
- Registrierte Dateien

In bestimmten Abständen werden alle Timestamps überprüft und alte Peers entfernt. Wenn für eine FileId keine Peers mehr vorhanden sind, wird sie gelöscht.

3.5.2 Peerato Plug-In

Das Peerato Plug-In muss die angebotenen Dateien, die fehlenden Chunks und die kennengelernten Peers verwalten. Der *DownloadManager* speichert die angebotenen Dateien und die kennengelernten Peers in einer Map. Mit der FileId kann der zugehörige *FileStore* erhalten werden. Peers werden mit ihrer IP-Socket-Adresse als Key identifiziert.

Fehlende Chunks

Die fehlenden Chunks werden durch ein Chunk-Objekt repräsentiert und von einer *ChunkSelectionStrategy* verwaltet. *RarestChunkFirst* ist eine mögliche Strategie und liefert den unter den anderen Peers seltensten Chunk, den ein gegebener Peer hat. Eine konkrete Strategie implementiert die folgenden Methoden:

addMissingChunks fügt die fehlenden Chunks des gegebenen FileStores zur Strategie hinzu.

getNextChunk gibt den nächsten Chunk zurück, der von einem gegebenen Peer angefordert werden soll.

removeChunk entfernt den Chunk mit der gegebenen ChunkId aus der Strategie.

Die ChunkId besteht aus der FileId und der Chunk Nummer. Ein Chunk-Objekt enthält eine Liste von Peers, die den Chunk haben, so dass schnell gezählt werden kann, wieviele Peers einen gegebenen Chunk zur Verfügung stellen.

Peers

Für jeden kennengelernten Nachbarn wird ein Peer-Objekt erstellt. Ein Peer referenziert diejenigen fehlenden Chunks, welche er bereits hat. So kann die RarestChunkFirst-Strategie den seltensten Chunk anfordern. Wird von einem Peer eine UploadOffer erhalten, iteriert die Strategie über alle Chunks des Peers und merkt sich den Chunk, den die wenigsten anderen Peers haben.

Peers, die einen DownloadRequest gesendet haben, werden von einer *PeerSelectionStrategy* verwaltet. Derzeit sind die *FirstInQueue*-Strategie und die *CreditQueue*-Strategie implementiert. Wird ein UploadSlot frei, sendet die FirstInQueue-Strategie dem am längsten wartenden Peer eine UploadOffer. Bei der CreditQueue-Strategie werden Peers, die mehr Chunks gesendet als empfangen haben, bevorzugt und müssen weniger lange warten.

FileStore

Ein FileStore wird für jede angebotene Datei erstellt, wobei auch Dateien als angeboten gelten, zu denen noch keine Chunks empfangen wurden. Er stellt mit *RandomAccessFile* eine Verbindung zu den Daten auf der Festplatte her. Beim Erstellen wird jeder Chunk mit seinem Hashwert überprüft und die fehlenden Chunks ermittelt, so dass die kompletten Chunks bei einem Neustart nicht nochmals heruntergeladen werden müssen. Mit *getChunkData* können zu einer gegebenen Chunk Nummer die Daten gelesen und dem anfragenden Peer geschickt werden. Mit *setChunkData* wird ein empfangener Chunk mit dem Hashwert überprüft und gespeichert.

HashFile

Das HashFile ist serialisierbar und wird aus einer XML-Datei gelesen, die den Namen *[Name der Datei].hashfile.xml* trägt. Folgende Daten sind darin enthalten:

- FileId (Hashwert der gesamte Datei)
- Grösse der Datei
- Grösse der Chunks
- Liste mit Hashwerten der Chunks
- verwendeter Hash-Algorithmus

Mit *HashFile.createHashFile* kann aus einer gegebenen Datei, der Angabe des Hash-Algorithmus' und der Grösse der Chunks ein HashFile erstellt werden.

3.5.3 Settings

Timeout Verbindung Eine Verbindung wird geschlossen, wenn für eine bestimmte Zeit keine Daten mehr gesendet oder empfangen wurden. Wenn dies eine ausgehende Datenverbindung war, wird der entsprechende Peer aus dem Upload-Slot entfernt. Das Timeout wird auch verwendet, damit angeforderte Chunks nach einer gewissen Zeit nochmals angefordert werden können, falls der anbietende Peer den Chunk aus irgend einem Grund nicht sendet. Der Grund könnte Überlastung des Senders, Ausfall des Senders, oder ein Gegner sein, der allen Peers UploadOffers anbietet, ohne danach etwas zu senden.

Upload-Geschwindigkeit Die Upload-Geschwindigkeit kann jeder Benutzer gemäss seiner eigenen Bandbreite einstellen. Dies ermöglicht auch anderen Anwendungen, einen Teil der Bandbreite nutzen zu können.

Anzahl Upload-Slots Die Anzahl der Upload-Slots hängt von der Upload-Geschwindigkeit ab. Ein Peer, der nur über eine Upload-Geschwindigkeit von 1 KB/sec verfügt, sollte die Anzahl der Upload-Slots auf 1 beschränken. So kann ein Chunk der Grösse 32 KB immer noch in weniger als einer Minute

Nachricht	Grösse in Bytes
PeerListRequest	200
PeerListAnswer	240
ChunkListRequest	170
ChunkListAnswer	200
DownloadRequest	100
UploadOffer	100
ChunkRequest	180
HaveChunk	180

Tabelle 3.1: Grössen der Nachrichten

geschickt werden, so dass kein Timeout beim Empfänger entstehen kann. Ein Peer mit höherer Upload-Geschwindigkeit sollte auch die Anzahl der Upload-Slots erhöhen. So kann der Peer kontinuierlich senden, auch wenn einer der Peers langsam downloadet, oder nach einer UploadOffer auf den ChunkRequest gewartet werden muss.

3.6 Overhead-Berechnungen

Das Downloaden mit einem File-Sharing System statt von einem zentralen Server ist mit einem Overhead verbunden. In Peerato verursacht nebem dem Nachrichtenaustausch mit dem Tracker und den anderen Peers auch das Downloaden des Hashfiles einen Overhead. Der Overhead durch das Hashfile hängt davon ab, in wieviele Chunks die Datei unterteilt wurde. Bei einer Anzahl von bis zu zehn Chunks ist das Hashfile ein paar hundert Bytes gross, danach steigt die Grösse proportional zur Anzahl von Chunks in den Kilobyte Bereich. Wir nehmen in den folgenden Berechnungen an, dass die Chunk-Grösse auf 32 KB festgelegt ist.

Der Nachrichten-Overhead nimmt mit der Anzahl von bekannten Peers zu, da dann jedem dieser Peers ein ChunkListRequest gesendet und ein ChunkList-Answer empfangen werden muss. Zusätzlich muss nach Empfang jedes Chunks ein HaveChunk an diejenigen bekannten Peers gesendet werden, die ihn noch nicht haben. Tabelle 3.1 zeigt die Grössen der einzelnen Nachrichten.

Definieren wir nun den Download-Overhead als den Anteil an zusätzlichen Daten, die für den Download der Datei empfangen werden müssen. 100% Overhead würde also bedeuten, dass zusätzlich zur Grösse der Datei nochmals so viele Daten empfangen werden müssen. Gleichermassen ist der Upload-Overhead der Anteil an zusätzlichen Daten, die für den Download der Datei gesendet werden müssen. Tabelle 3.2 zeigt den Download- und Upload-Overhead für verschiedene Dateigrössen, wenn der Peer nur mit einem anderen Peer verbunden ist. Tabelle 3.3 zeigt die Overheads, wenn der Peer mit zehn anderen Peers verbunden ist. Mit zehn Peers ist der Overhead stets höher. Zum Kennenlernen wird jedem Peer ein ChunkListRequest gesendet und eine ChunkListAnswer empfangen. Dazu kommt bei jedem empfangenen Chunk noch ein Overhead für HaveChunk-Nachrichten hinzu, die an alle bekannten Peers gesendet werden, die den Chunk

Dateigrösse	Download-Overhead	Upload-Overhead
1 KB	109%	81%
10 KB	11%	8%
100 KB	2%	2%
1 MB	1%	1%

Tabelle 3.2: Download- und Upload-Overhead mit einem bekannten Peer

Dateigrösse	Download-Overhead	Upload-Overhead
1 KB	442%	476%
10 KB	44%	48%
100 KB	10%	11%
1 MB	6%	6%

Tabelle 3.3: Download- und Upload-Overhead mit zehn bekannten Peers

noch benötigen.

Für kleine Dateien ergibt sich – besonders wenn der Peer mit mehreren Peers verbunden ist – ein sehr grosser Overhead. Es lohnt sich daher kaum, eine 1 KB grosse Datei mit Peerato zu verteilen, da ein zentraler Server bereits mit einer Upload-Bandbreite von 100 kBit/s etwa 10 Clients pro Sekunde bedienen kann. Im Vergleich dazu kann auch ein zentraler Tracker nur etwa 50 Clients pro Sekunde bedienen, je nach Leistungsfähigkeit des Rechners.

Für Dateien ab 10 KB lohnt sich die Benutzung von Peerato, wenn sehr viele Peers die Datei gleichzeitig haben wollen, da sie zwar den Overhead spüren, die Datei jedoch schneller downloaden können als von einem überlasteten Server. Bei Dateien ab 100 KB ist der Overhead kaum noch spürbar, weshalb Peerato für diese Dateien gut geeignet ist.

3.7 Integration in Spamato

Spamato ist ein flexibles Plug-In System mit einem Update-Mechanismus. Bisher konnten die Plug-Ins von einem HTTP-Server oder vom lokalen Profil heruntergeladen werden. Für neue Protokolle gab es einen Extension-Point, so dass neben dem HttpUpdateHandler noch weitere UpdateHandler hinzugefügt werden konnten. Das Suchen und Downloaden von Plug-Ins erfolgte dann mit dem entsprechenden UpdateHandler. Der HTTP-Server musste eine Datei *versions.txt* bereitstellen, in der jeweils die aktuellste Version eines Plug-Ins verzeichnet war. Die Anpassung der Datei erfolgte manuell. Abbildung 3.5 verschafft einen Überblick über den bisherigen Update-Mechanismus.

Jedoch muss das Suchen und Downloaden weder mit demselben Protokoll noch mit demselben Server erfolgen. Deshalb wird neu der UpdateHandler in einen SearchHandler mit Extension-Point „search.protocol“ und einen DownloadHandler mit Extension-Point „download.protocol“ unterteilt.

Der SearchHandler fragt den *PublishServer* nach registrierten Versionen der Plug-Ins. Dieser teilt ihm dann mit, von wo die Plug-Ins heruntergeladen werden können. Der PublishServer kann ein normaler HTTP-Server sein, muss aber

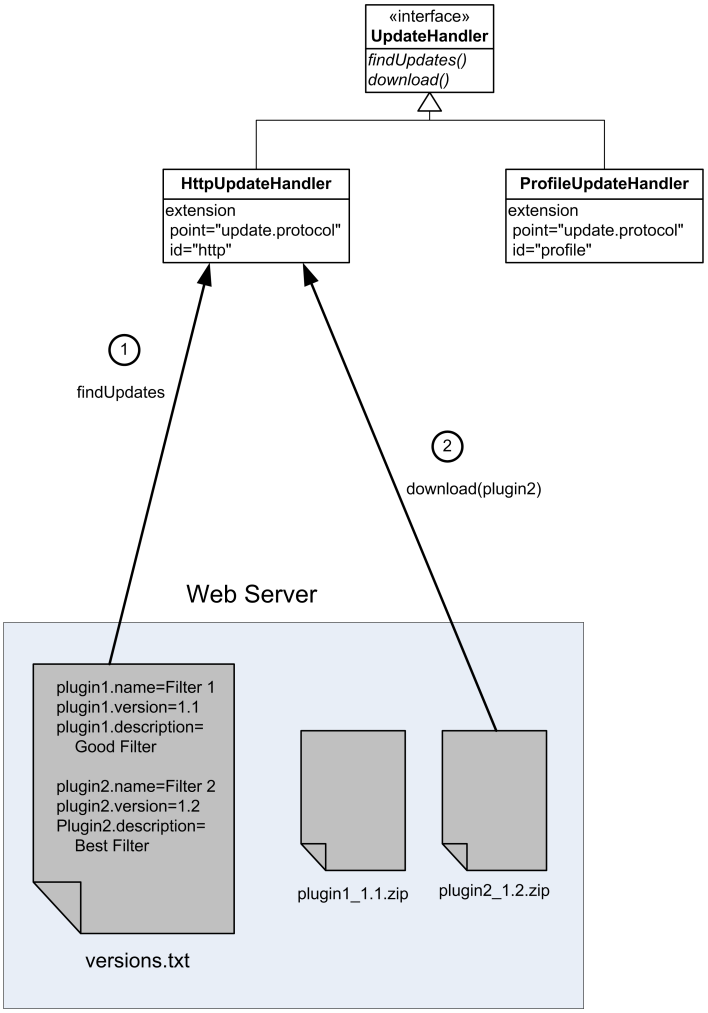


Abbildung 3.5: Bisheriger Update-Mechanismus in Spamato. Der HttpUpdate-Handler sucht zuerst die *versions.txt*-Datei auf dem Web-Server. Dann weiss er, welche Plug-In Versionen vorhanden sind und kann diese downloaden und installieren.

keine Plug-Ins speichern. Die Download-URLs der Plug-Ins können auf beliebige Server mit beliebigen Protokollen verweisen, wie HTTP, FTP, Peerato oder File. Diese Protokolle werden dann vom entsprechenden DownloadHandler richtig verarbeitet.

Der HttpDownloadHandler hat die gleiche Funktion wie die Methode *download* im ursprünglichen HttpUpdateHandler. Der PeerDownloadHandler verarbeitet das Peerato-Protokoll. Die Download-URL verweist auf einen *HashFileServer*. Der Handler lädt das HashFile herunter und ruft Peerato auf, um das Plug-In von anderen Peers zu erhalten. Abbildung 3.6 zeigt einen Überblick zu Search-Handlern und DownloadHandlern. Im ersten Schritt teilt der PublishServer die Versionen mit den Download-URLs mit. Dann wird je ein Plug-In von einem HTTP-Server und von Peerato heruntergeladen.

Das Bereitstellen eines Updates oder eines neuen Plug-Ins erfolgt nun automatisch. Mit der Spamato-WebConfig (siehe Abbildung 3.7) wählt der Entwickler die Zip-Datei seines Plug-Ins und den zu verwendenden Plug-In Key. Danach legt er noch die Publish- und die Download-Server des Plug-Ins fest. Version, Name und Beschreibung des Plug-Ins werden neu automatisch aus dem *plugin.xml* in der Zip-Datei gelesen. Danach werden die Plug-Ins an die Download-Server geschickt und diese URLs zusammen mit den anderen Informationen über das Plug-In an die Publish-Server geschickt.

Um die verschiedenen Download-Protokolle zu unterstützen, wird für den Upload der UploadHandler mit dem Extension-Point „upload.protocol“ aufgerufen. Der PeerUploadHandler erstellt dann ein HashFile aus der Datei und sendet dieses an den HashFile-Server. Die Zip-Datei selbst wird mit Peerato anderen Benutzern zum Download angeboten.

Ein PublishHandler sorgt dann dafür, dass die entsprechende URL auf dem Publish-Server veröffentlicht wird. Da es denkbar ist, dass mehrere Entwickler am selben Plug-In arbeiten und jeder sein eigenes Update bereitstellen will, kann auf einem Publish-Server die gleiche Version mit verschiedenen URLs vorhanden sein.

3.7.1 Sicherheit

Da jeder Entwickler ein Plug-In bereitstellen kann, muss es dem Benutzer erleichtert werden, zwischen sicheren und unsicheren Plug-Ins zu unterscheiden. Updates können wie bisher nur auf der in der Plug-In Beschreibung verzeichneten Update-URL gesucht werden, oder von einem separat hinzugefügten Update-Server. Denkbar ist ein firmeninterner Update-Server, dem der Benutzer vertraut. Will der Benutzer trotzdem eine Version eines anderen Servers installieren, muss er zuerst das vorhandene Plug-In deinstallieren.

Download-Informationen zu neuen Plug-Ins hingegen können von beliebigen Servern stammen. Informationen zu sicheren Plug-Ins sind auf einem Publish-Server gespeichert, auf dem nur bestimmte Entwickler Plug-Ins veröffentlichen können. Deshalb können für die Publish-Server IP-Bereiche definiert werden. Jedem IP-Bereich ist eine andere *versions.xml*-Datei zugewiesen. Wenn ein Entwickler sein Plug-In veröffentlicht, wird seine IP-Adresse überprüft und die Informationen in der entsprechenden *versions.xml*-Datei gespeichert.

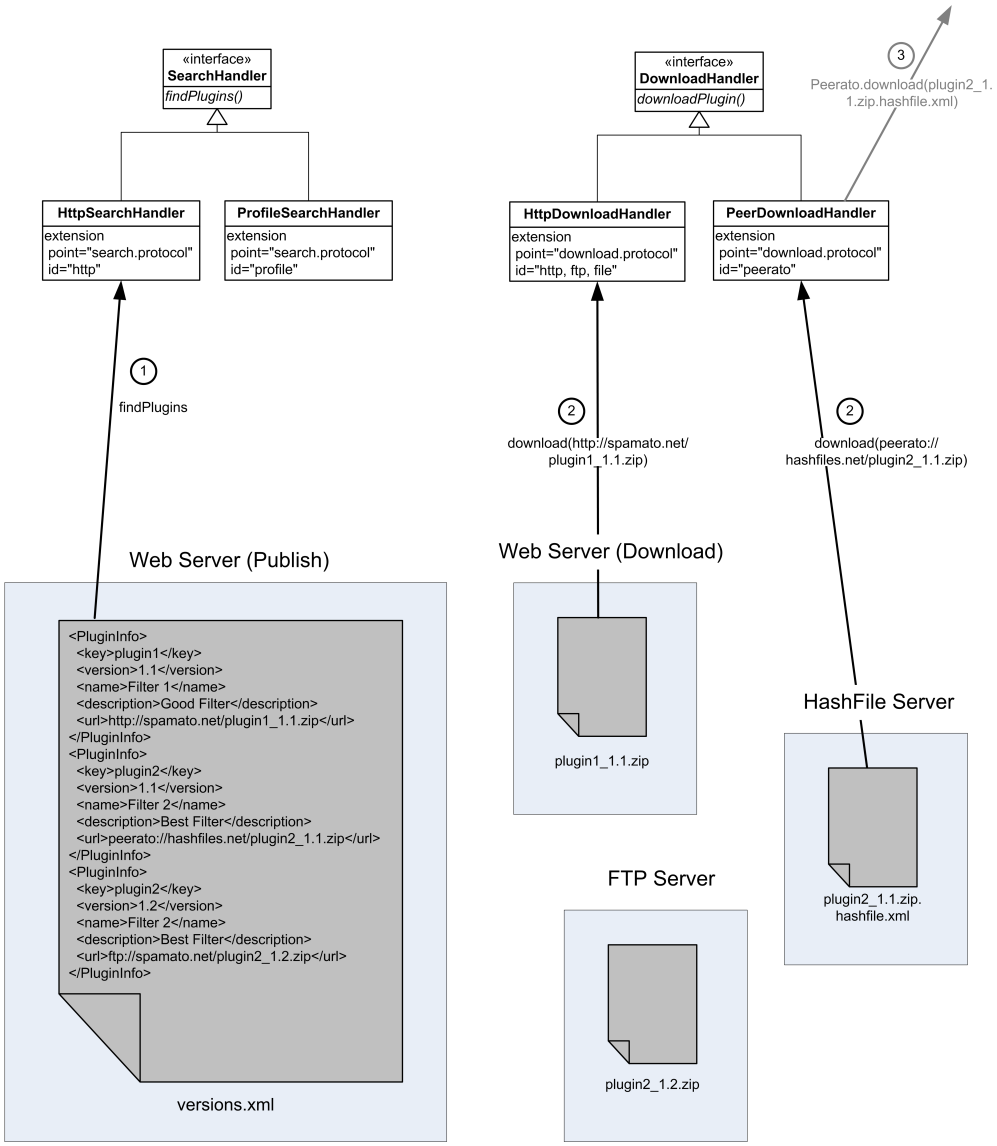


Abbildung 3.6: Neuer Update-Mechanismus in Spamato. Der HttpSearchHandler sucht zuerst die versions.xml-Datei auf dem Web-Server. Dann weiss er, welche Plug-In Versionen vorhanden sind und kann den entsprechenden DownloadHandler aufrufen, um welche zu downloaden und zu installieren.

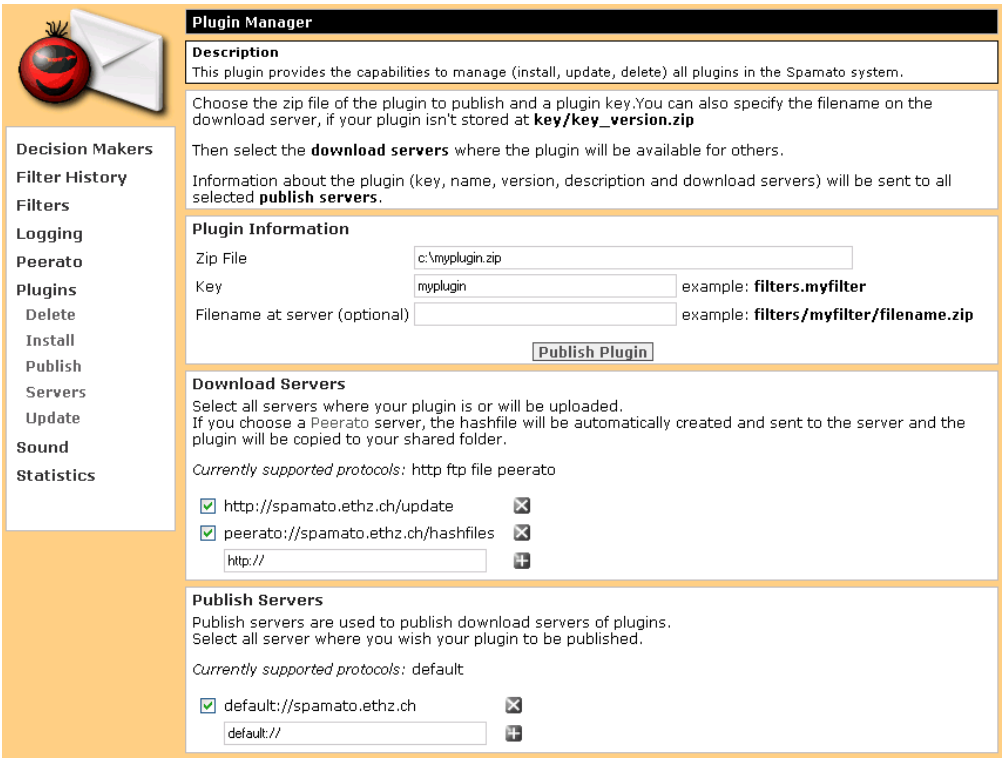


Abbildung 3.7: Screenshot: Bereitstellen eines neuen Plug-Ins.

Konkret werden Veröffentlichungen vom SPAMATO-Entwickler-Team in der Datei */update/versions.xml* gespeichert, während beliebige Entwickler ihre Plug-Ins in der Datei */openpublish/versions.xml* veröffentlichen. Ein Benutzer hat dann bei sich zwei Install-Server verzeichnet, *http://spamato.ethz.ch/update* und *http://spamato.ethz.ch/openpublish*. Ersteren kreuzt er als vertrauenswürdig an, da dort nur Entwickler vom SPAMATO-Team veröffentlichen können. Letzterer ist zugänglich für alle und deshalb nicht vertrauenswürdig. Installiert er ein Plug-In von einem Server, der nicht vertrauenswürdig ist, wird vor der Installation eine Warnmeldung angezeigt und nachgefragt, ob er es wirklich installieren will.

4 Neighbor Selection Strategien

In den meisten P2P-Verteilsystemen werden die Dateien in Chunks aufgeteilt. So kann ein Peer einen fertigen Chunk bereits an andere Peers verteilen, ohne die ganze Datei zu haben.

Das Hauptproblem in solch einem System ist es, die Peers so untereinander zu verbinden, dass möglichst schnell alle Peers alle Chunks haben. Ebenfalls eine wichtige Rolle spielt dabei die Wahl des Chunks, der zu einem bestimmten Zeitpunkt von einem Peer zu einem anderen gesendet wird.

In [QS04] wurde ein Modell für das BitTorrent Netzwerk entwickelt und im Steady-State analysiert. Im Steady-State kann man annehmen, dass die Chunks im Netz gleichverteilt sind, sowohl global, als auch lokal gesehen. Unter diesen Annahmen ist die verfügbare Upload-Bandbreite fast optimal genutzt. Besonders für kleinere Dateien ist es aber interessant, wie sich die Chunks im Netz ausbreiten – beispielsweise ob und unter welchen Umständen es dazu kommt, dass die Chunks im Steady-State gleichverteilt sind.

Für Anwendungen mit einer automatischen Update-Komponente ist es auch interessant zu sehen, wie das System auf die plötzliche Popularität einer Datei reagiert. In Messungen von [PGES05] wird eine Datei der Grösse 1.87 GByte erwähnt, bei der es in den ersten fünf Tagen kein BitTorrent-Benutzer geschafft hat, die komplette Datei zu erhalten.

Um die Verbreitungseffizienz verschiedener Topologien zu vergleichen, müssen mehrere Grössen angeschaut werden. Eine mögliche Grösse ist die längste Downloadzeit eines Peers. Die durchschnittliche Downloadzeit kann aus der gesamthaft genutzten Upload-Geschwindigkeit berechnet werden. Bei der Verteilung spielt auch der durchschnittliche und der längste Weg zum Seeder sowie der durchschnittliche und der maximale Knotengrad eine Rolle.

4.1 Modelle

Wir werden die verschiedenen Topologien zuerst anhand eines einfachen Modells vergleichen und dann einige Aspekte eines komplexeren Modells anschauen.

4.1.1 Einfaches Modell

Ankunft der Peers Die Peers kommen alle fast gleichzeitig an. Die Topologie der Peers ist also bereits aufgebaut, bevor der erste Chunk verschickt wird. Diese Annahme macht Sinn, wenn die Anwendung eine automatische Update-Komponente (Push-Prinzip) beinhaltet. Dies ist der schlechteste Fall, da dann die Chunks noch nicht im Netz verteilt sind. Später ankommende Peers können von bisherigen Peers schneller bedient werden, wenn schon Chunks im Netz sind.

Ausfall der Peers Die Peers bleiben solange online, bis alle anderen Peers die Datei erhalten haben. In einer Anwendung wie SPAMATO ist diese Annahme berechtigt, da einem Peer durch das Anbieten der Datei keine grossen Nachteile (Kosten) entstehen. Die Dateien sind klein und die Inhalte nicht illegal. Ausserdem sind Downloads selten, ausser wenn das Plug-In gerade neu herausgekommen ist.

Bandbreiten der Peers Alle Peers haben dieselbe Upload-Bandbreite. Die Download-Bandbreite ist nicht begrenzt.

4.1.2 Komplexeres Modell

Ankunft der Peers Die Peers kommen mit einer bestimmten Ankunftsrate an. Wir nehmen wie [QS04] an, dass diese einem Poisson-Prozess folgt. Die Anzahl der Peers N ist nicht begrenzt, das System sollte sowohl mit wenigen als auch mit sehr vielen Peers zurechtkommen.

Ausfall der Peers Die Peers fallen mit geringer Wahrscheinlichkeit während dem Downloaden aus. Nach dem Downloaden stellen sie ihre Upload-Bandbreite noch exponentialverteilt lange zur Verfügung oder fallen sogar sofort aus. Hingegen fällt der Seeder nie aus, da er ja will, dass möglichst viele sein Plug-In ausprobieren.

Bandbreiten der Peers Peers können unterschiedliche Bandbreiten haben. Analog-Modem-Benutzer haben eine geringe Upload- und Download-Bandbreite, während Cable/DSL-Modem-Benutzer höhere Bandbreiten erreichen. Die Download-Bandbreite eines Peers ist höher als die Upload-Bandbreite. Einzelne Benutzer sind Freerider und stellen gar keine Upload-Bandbreite zur Verfügung.

Anzahl Chunks Bei der Wahl der Anzahl von Chunks C muss der Tradeoff zwischen Parallelität und Overhead beachtet werden. Wird die Datei in wenige grosse Chunks aufgeteilt, hat man eine kleine Parallelität, da es lange dauert, bis ein Peer einen Chunk heruntergeladen hat und weiterverteilen kann. Hat man für jeden Chunk einen konstanten Overhead, wie zum Beispiel Verbindungsaufbau oder Hashwert, sollte die Datei jedoch nicht in zu viele kleine Chunks aufgeteilt werden. Wir nehmen deshalb an, dass die Anzahl der Chunks beliebig ist. Bei einer ganz kleinen Datei ist es auch möglich, dass diese nur aus einem Chunk besteht.

Anzahl Verbindungen Ein Peer kann sich nur mit einer begrenzten Anzahl von Peers verbinden und hat damit nur lokales Wissen.

4.2 Lower Bound

Schauen wir uns die Lower Bound unter dem einfachen Modell mit gleichen Upload-Bandbreiten und unbegrenzten Download-Bandbreiten an. Ein optimaler Algorithmus mit globaler Sicht würde dafür sorgen, dass alle Peers so schnell

wie möglich einen Chunk haben, damit möglichst viel der gesamten Upload-Bandbreite genutzt werden kann.

Theorem. *Das Verteilen von C Chunks an N Peers dauert bei einer Bandbreite von einem Chunk pro Runde mindestens $\lceil \log_2 N \rceil + C - 1$ Runden.*

Beweis. Es dauert mindestens $C - 1$ Runden, bis der Seeder alle bis auf den letzten Chunk an einen Peer weitergegeben hat. Der letzte Chunk muss nun noch an N Peers verteilt werden. In der i-ten Runde können höchstens 2^{i-1} Peers den Chunk uploaden, da sich die Besitzer des Chunks mit jeder Runde höchstens verdoppeln können. Bei der letzten Runde haben $2^{x-1} = \frac{N}{2}$ Peers den Chunk und können an die anderen $\frac{N}{2}$ Peers uploaden. Also dauert das Verteilen des letzten Chunk mindestens $x = \lceil \log_2 N \rceil$ Runden, und somit das Verteilen aller Chunks $\lceil \log_2 N \rceil + C - 1$ Runden. □

4.2.1 Heterogene Bandbreiten

Haben die Peers unterschiedliche Upload-Bandbreiten und weiterhin unbeschränkte Download-Bandbreiten, kann in zwei Grössen optimiert werden. Einerseits kann die Download-Zeit aller Peers minimiert werden. Andererseits ist es fair, wenn Peers mit höherer Upload-Bandbreite auch schneller fertig sind. Man kann also auch die Download-Zeit umgekehrt proportional zur jeweiligen Upload-Bandbreite des Peers minimieren.

Nehmen wir an, dass zu Beginn der Verteilung ein oder mehrere Seeder im System sind, die eine gesamte Upload-Bandbreite von B Chunks pro Runde haben. Weiter habe jeder Peer als Upload-Bandbreite ein Vielfaches von einem Chunk pro Runde.

Die Idee ist nun, dass die Peers in Gruppen aufgeteilt werden, so dass die gesamte Upload-Bandbreite einer Gruppe $\geq B$ Chunks pro Runde beträgt. So können die Gruppen in einer ersten Phase wie einzelne Peers mit homogener Bandbreite behandelt werden. Eine zweite Phase wird dann noch benötigt, um die Chunks an alle Peers innerhalb der Gruppe zu verteilen. Wir entwickeln nun die Lower Bounds für die Verteilung mit heterogenen Bandbreiten, unter der Voraussetzung, dass die Verteilung mit Gruppen erfolgt.

Phase 1

In Phase 1 sendet eine Gruppe jeweils B verschiedene Chunks an eine andere Gruppe. Ein Peer i mit Bandbreite B_i erhält B_i Chunks, für deren Weiterverteilung an andere Gruppen er verantwortlich ist. Phase 1 endet, wenn alle Gruppen alle Chunks haben.

Abbildung 4.1 zeigt einen Seeder mit Bandbreite 6 und Gruppen mit Peers mit heterogenen Bandbreiten. Jede Gruppe hat eine Upload-Bandbreite von 6 Chunks pro Runde. Die Abbildung zeigt zudem mögliche Verbindungen mit den Chunks, die in einer Sende-Runde der entsprechenden Gruppen über diese Verbindung fliessen. Jeder Peer erhält in einer Runde soviele Chunks von einer anderen Gruppe, wie er pro Runde senden kann.

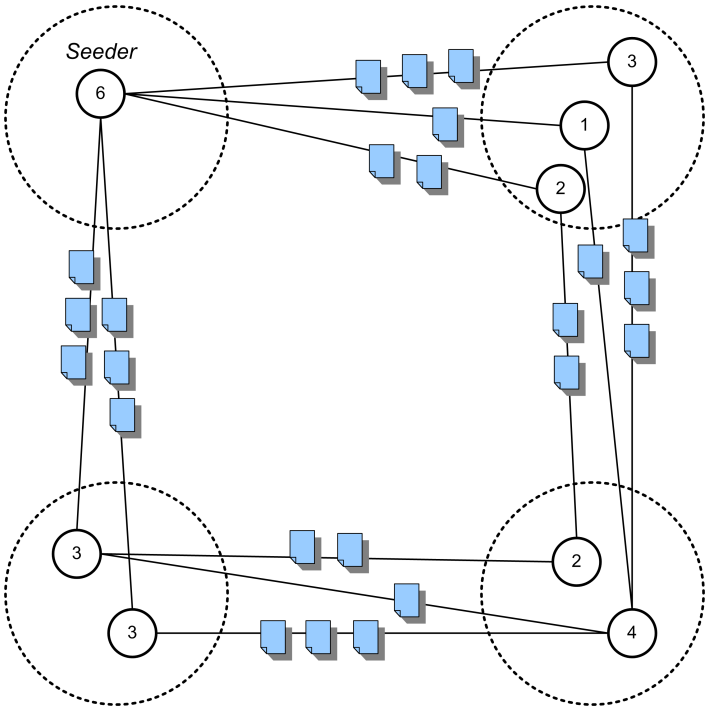


Abbildung 4.1: Heterogene Verteilung Phase 1: Aufteilung in Gruppen mit Bandbreite $B = 6$ mit möglichen Verbindungen zwischen den Peers. Die Chunks auf einer Verbindung deuten an, wieviele Chunks in einer Runde darüber fliessen, wenn die eine Gruppe an die andere sendet.

Lemma. *Phase 1 dauert mit G Gruppen mindestens $\lceil \log_2 G \rceil + \frac{C}{B} - 1$ Runden.*

Beweis. Um alle bis auf B Chunks weiterzugeben, benötigt ein Seeder mit Bandbreite B Chunks pro Runde

$$(C - B)\,\frac{1}{B} = \frac{C}{B} - 1$$

Runden. Die B letzen Chunks müssen nun noch an G Gruppen verteilt werden, was mindestens $\lceil \log_2 G \rceil$ Runden dauert. □

Die Download-Bandbreite von Phase 1 hängt vom konkreten Algorithmus zum Verteilen der Chunks an die Gruppen ab. Jedoch muss Peer i in den Runden, in denen die Gruppe etwas erhält, mindestens B_i Chunks empfangen können.

Phase 2

In Phase 2 werden die Chunks innerhalb der Gruppe verteilt. Abbildung 4.2 zeigt 3 Peers einer Gruppe mit 6 Chunks, die sie in einer Runde von Phase 1 erhalten haben. Phase 2 verteilt diese Chunks in $n - 1 = 2$ Runden innerhalb der Gruppe.

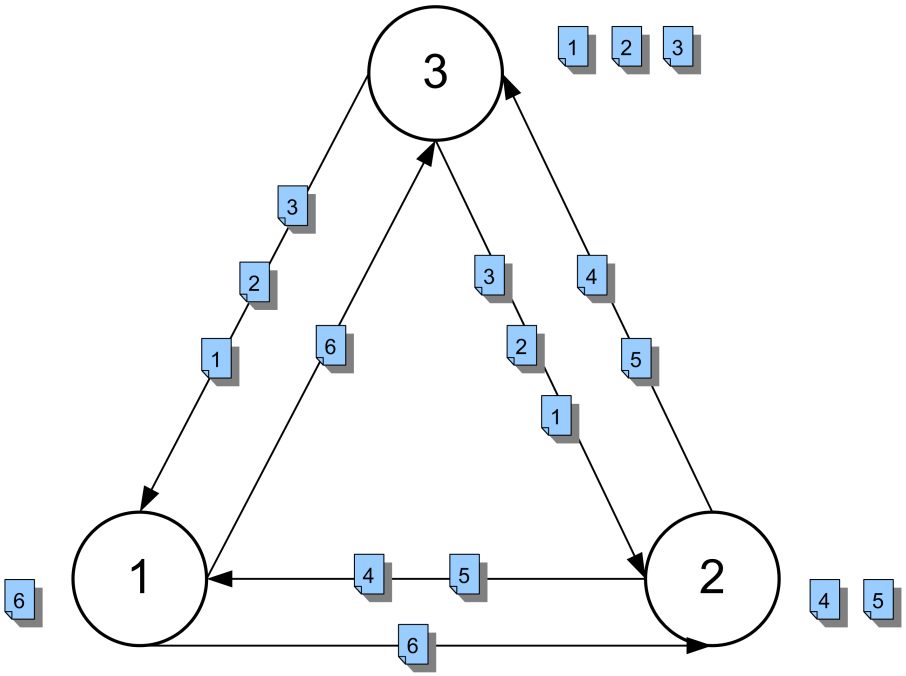


Abbildung 4.2: Heterogene Verteilung Phase 2: Gruppe mit $n = 3$ Peers und Bandbreite $B = 6$.

Lemma. *Phase 2 dauert in einer Gruppe mit n Peers mindestens* $(n-1)\frac{C}{B}$ *Runden.*

Beweis. Zu Beginn der Phase 2 besitzt Peer i entsprechend seiner Bandbreite $\frac{B_i}{B}$ aller Chunks, also $C\frac{B_i}{B}$ Chunks. Das Verteilen dieser Chunks an die $n-1$ anderen Peers benötigt

$$(n-1)\,C\frac{B_i}{B}\frac{1}{B_i} = (n-1)\,\frac{C}{B}$$

Runden, da er in einer Runde B_i Chunks versenden kann. □

Lemma. *In Phase 2 beträgt die erforderliche Download-Bandbreite von Peer i in einer Gruppe mit n Peers mindestens* $\frac{B-B_i}{n-1}$*, wenn die Phase* $(n-1)\frac{C}{B}$ *Runden dauert.*

Beweis. Zu Beginn der Phase 2 fehlen dem Peer $C\left(1-\frac{B_i}{B}\right)$ Chunks, die in $(n-1)\frac{C}{B}$ Runden empfangen werden müssen. Die Anzahl der Chunks pro Runde beträgt daher:

$$C\left(1-\frac{B_i}{B}\right)/\,(n-1)\,\frac{C}{B} = C\left(1-\frac{B_i}{B}\right)\frac{B}{C}/\,(n-1) = \frac{B-B_i}{n-1}$$

□

Die Dauer von Phase 2 hängt davon ab, wieviele Peers in einer Gruppe sind. Wir untersuchen in den folgenden beiden Abschnitten die Dauer der gesamten Verteilung, wenn jede Gruppe gleichviele Peers oder Peers mit gleicher Bandbreite enthält.

Gruppen mit gleichvielen Peers

Um die Download-Zeit aller Peers zu minimieren, enthält jede Gruppe vorzugsweise gleichviele Peers.

Theorem. *Werden N Peers mit unterschiedlichen Bandbreiten* B_i *so in Gruppen aufgeteilt, dass jede Gruppe genau n Peers enthält, und* $\sum_{i=1}^{n} B_i = B$*, so beträgt die Anzahl der benötigten Runden zur Verteilung von C Chunks mindestens*

$$\left\lceil \log_2 \frac{N}{n} \right\rceil + \frac{n}{B}C - 1$$

Beweis. Da die Anzahl der Gruppen $G = \frac{N}{n}$ ist, dauert Phase 1 $\left\lceil \log_2 \frac{N}{n} \right\rceil + (C-B)\,\frac{1}{B}$ Runden und Phase 2 $(n-1)\frac{C}{B}$ Runden, gesamthaft also:

$$\begin{aligned} & \left\lceil \log_2 \frac{N}{n} \right\rceil + \frac{C}{B} - 1 + (n-1)\,\frac{C}{B} \\ = & \left\lceil \log_2 \frac{N}{n} \right\rceil + (C + nC - C)\,\frac{1}{B} - 1 \\ = & \left\lceil \log_2 \frac{N}{n} \right\rceil + \frac{n}{B}C - 1 \end{aligned}$$

□

Für $B = 1$ und $n = 1$ erhält man die Lower Bound für homogene Upload-Bandbreiten. Wenn der Seeder Bandbreite $B = 6$ hat, alle anderen aber $B_i = 1$ und somit $n = 6$, so verbessert sich lediglich der logarithmische Teil der Dauer gegenüber der homogenen Verteilung. $n\frac{C}{B} - 1 = C - 1$ entspricht dann genau dem zweiten Teil der Dauer der homogenen Verteilung. Wenn hingegen auch andere Peers höhere Upload-Bandbreiten haben, verbessert sich auch der zweite Teil. Ist wieder $B = 6$, aber $n = 3$, so dauert der zweite Teil $\frac{n}{B}C - 1 = \frac{C}{2} - 1$ und ist somit etwa doppelt so schnell wie die homogene Verteilung. Abbildung 4.3(a) zeigt Gruppen mit jeweils $n = 3$ Peers mit gesamter Upload-Bandbreite $B = 6$.

Ein Nachteil dieser Aufteilung in Gruppen ist, dass in Phase 2 ein Peer mit kleiner Upload-Bandbreite eine umso grössere Download-Bandbreite benötigt. Ist wieder $B = 6$ und $n = 3$, so benötigt ein Peer mit Upload-Bandbreite $B_i = 1$ eine Download-Bandbreite von $\frac{B-B_i}{n-1} = \frac{6-1}{3-1} = 2.5$ Chunks pro Runde.

Gruppen mit Peers gleicher Bandbreite

Teilt man die Peers so in Gruppen auf, dass alle Peers einer Gruppe die gleiche Upload-Bandbreite haben, werden Peers mit höherer Upload-Bandbreite schneller fertig.

Theorem. *Werden N Peers mit unterschiedlichen Bandbreiten so in Gruppen aufgeteilt, dass in Gruppe i $\frac{B}{B_i}$ Peers gleicher Bandbreite B_i sind, so dauert die Verteilung von C Chunks mindestens*

$$\lceil \log_2 G \rceil + \frac{C}{B_i} - 1$$

Runden.

Beweis. Phase 1 dauert wie gewohnt $\lceil \log_2 G \rceil + \frac{C}{B} - 1$ Runden. Die Anzahl der Peers in Gruppe i ist $n = \frac{B}{B_i}$, also dauert Phase 2 $\left(\frac{B}{B_i} - 1\right)\frac{C}{B}$ Runden. Die Anzahl der benötigten Runden für die gesamte Verteilung beträgt also:

$$\begin{aligned} & \lceil \log_2 G \rceil + \frac{C}{B} - 1 + \left(\frac{B}{B_i} - 1\right)\frac{C}{B} \\ = \; & \lceil \log_2 G \rceil + \frac{C}{B} - 1 + \frac{C}{B_i} - \frac{C}{B} \\ = \; & \lceil \log_2 G \rceil + \frac{C}{B_i} - 1 \end{aligned}$$

□

Abgesehen vom logarithmischen Teil benötigt also ein Peer i mit Upload-Bandbreite $B_i = 2$ nur halb so viel Zeit wie ein Peer j mit Upload-Bandbreite $B_j = 1$ und damit auch doppelt so schnell wie wenn man die homogene Verteilung anwenden würde.

Ein Vorteil im Vergleich zur Aufteilung in Gruppen mit gleichvielen Peers ist, dass in Phase 2 die benötigte Download-Bandbreite die Upload-Bandbreite nicht übersteigt.

Theorem. *In Phase 2 beträgt die erforderliche Download-Bandbreite von Peer i mindestens B_i.*

Beweis. Die erforderliche Download-Bandbreite in Phase 2 ist $\frac{B-B_i}{n-1}$. Für $n = \frac{B}{B_i}$ erhalten wir:

$$\frac{B - B_i}{n-1} = \frac{B - B_i}{\frac{B}{B_i} - 1} = \frac{(B - B_i)\, B_i}{B - B_i} = B_i$$

□

Abbildung 4.3 zeigt einen Vergleich der Aufteilung in Gruppen. Beide Beispiele haben einen Seeder mit Bandbreite $B = 6$. Die Gruppen enthalten in Abbildung 4.3(a) jeweils $n = 3$ Peers, in Abbildung 4.3(b) Peers gleicher Bandbreiten.

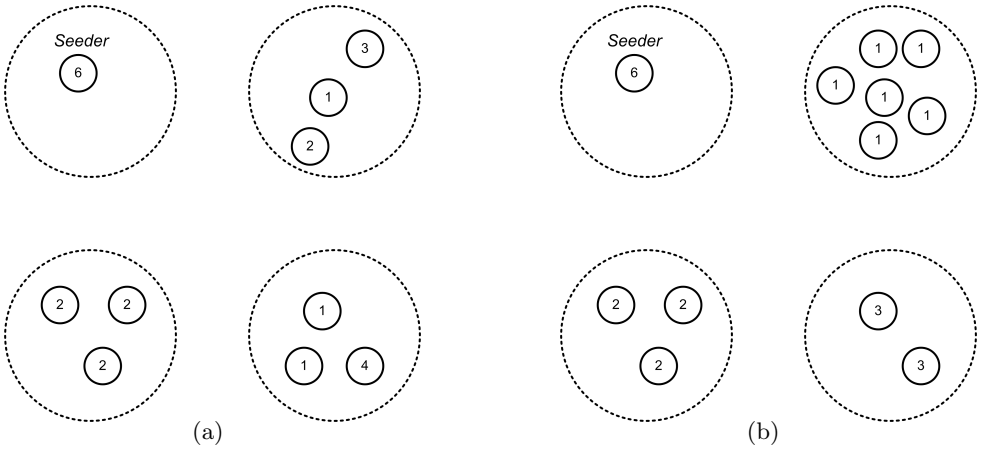


Abbildung 4.3: (a) Gruppen mit gleichvielen Peers, (b) Gruppen mit gleichschnellen Peers

4.3 Topologien

4.3.1 Baum

Werden die Peers baumartig miteinander verbunden, sendet ein innerer Knoten jeden Chunk an alle seine Nachfolger. Jeder Peer hat nur einen Vorgänger, der ihm Chunks senden kann.

Theorem. *In einem binären Baum dauert das Verteilen von C Chunks an $N-1$ Peers $2\left(\lfloor \log_2 N \rfloor + C - 1\right)$ Runden, wenn die Uploadrate homogen ist und 1 Chunk pro Runde beträgt.*

Beweis. In einem binären Baum muss ein innerer Knoten jeden Chunk zweimal uploaden, da er zwei Nachfolger hat, die nur von ihm Chunks bekommen können. Bei einer homogenen Uploadrate von 1 Chunk pro Runde werden daher $2\,(C-1)$ Runden benötigt, bis der Seeder alle bis auf den letzten Chunk an seine Nachfolger weitergegeben hat. Der letzte Chunk muss nun noch von der Wurzel

bis in alle Blätter verteilt werden. Ist N die Anzahl der Peers inklusive Seeder, so hat der Baum die Höhe $\lfloor \log_2 N \rfloor$. Die Verteilung des letzten Chunks dauert also $2 \lfloor \log_2 N \rfloor$ Runden. Gesamthaft sind somit $2 (\lfloor \log_2 N \rfloor + C - 1)$ Runden erforderlich. □

Es werden also doppelt so viele Runden benötigt wie bei einer optimalen Verteilung. Dies kann man sich damit erklären, dass stets höchstens die Hälfte der verfügbaren Upload-Bandbreite verwendet wird, da die Blätter des Baums niemandem uploaden können.

Theorem. *In einem Baum der Ordnung $b > 1$ dauert das Verteilen von C Chunks an $N - 1$ Peers $b (\lceil \log_b (N (b-1) + 1) \rceil - 1 + C - 1)$ Runden.*

Beweis. Ein Baum der Ordnung b mit Höhe h hat maximal $N = \frac{b^{h+1}-1}{b-1}$ Knoten. Die Höhe beträgt daher $\lceil \log_b (N (b-1) + 1) \rceil - 1$. Für die Verteilung der C Chunks werden deshalb $b (\lceil \log_b (N (b-1) + 1) \rceil - 1 + C - 1)$ Runden benötigt. □

Interessant ist der Fall $b = N - 1$, dem Downloaden von einem zentralen Server:

$$\begin{aligned}
& (N-1) \left(\left\lceil \log_{N-1} (N (N-1-1) + 1) \right\rceil - 1 + C - 1 \right) \\
= \; & (N-1) \left(\left\lceil \log_{N-1} (N-1)^2 \right\rceil - 1 + C - 1 \right) \\
= \; & (N-1) \left(2 \left\lceil \log_{N-1} (N-1) \right\rceil - 1 + C - 1 \right) \\
= \; & (N-1) (2 \lceil 1 \rceil - 1 + C - 1) \\
= \; & (N-1) C
\end{aligned}$$

Mit $b = 1$ hat jeder Knoten nur einen Nachfolger. Dies entspricht der Verteilung entlang eines linearen Pfades, was $N - 1 + C - 1$ Runden dauert. $C - 1$ Runden werden benötigt, bis die Wurzel alle bis auf einen Chunk verteilt hat, und $N - 1$ bis der letzte Chunk den letzten Knoten erreicht hat.

Vergleichen wir die Verteilung einer ganzen Datei mit der Aufteilung der Datei in C Chunks und nehmen dazu den binären Baum. Das Senden der gesamten Datei an 2 Nachfolger dauert $2C$ Runden. Da die Höhe des Baums $\lfloor \log_2 N \rfloor$ beträgt, dauert es $2C \lfloor \log_2 N \rfloor$ Runden, bis die Blätter die Datei erhalten haben. Da $\lfloor \log_2 N \rfloor \geq 1$ und $C \geq 1$, ist die Aufteilung in C Chunks immer gleich gut oder besser, als wenn die ganze Datei übertragen würde. Mit $N = 8$ Peers ist die Verteilung bei einer Aufteilung in $C = 4$ Chunks bereits doppelt so schnell.

Bäume sind sehr anfällig auf Ausfälle. Wenn ein innerer Knoten ausfällt, erhält der gesamte Teilbaum des Knotens keine Chunks mehr. Mögliche Verbesserungen beinhalten das Füllen innerer Knoten mit mehreren Peers. Für den Fall, dass ein gesamter Knoten ausfällt, können die Knoten noch zusätzlich mit dem Vorgängerknoten verbunden werden. Abbildung 4.4 zeigt einen Baum mit zusätzlichen Verbindungen. Wenn Knoten Nummer 2 ausfällt, können Chunks immer noch über die zusätzlichen Verbindungen fliessen. Ein zusätzlicher Ausfall des Knoten Nummer 6 könnte allerdings nicht mehr so gut bewältigt werden,

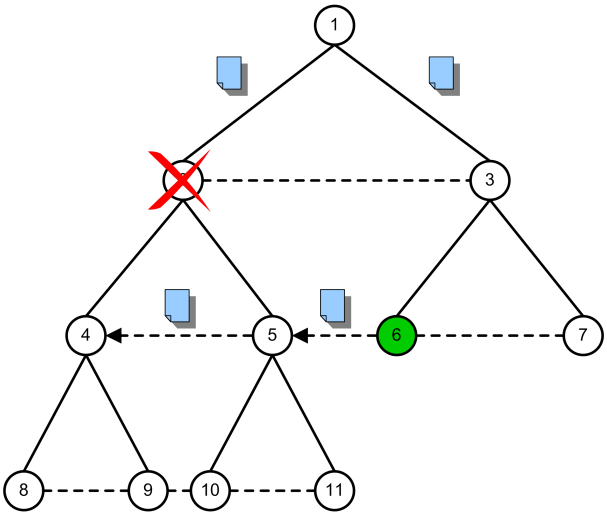


Abbildung 4.4: Baum: Innerer Knoten fällt aus.

da die Chunks dann über mehrere Ebenen hinunter- und wieder hinauffliessen müssen.

Wenn Peers sofort nach dem kompletten Downloaden ausfallen, kann ein Baum nicht verwendet werden, da ein innerer Knoten den letzten Chunk, den er erhalten hat, nicht mehr uploaden wird. Damit wird der Fluss der Chunks zu den Blättern unterbrochen. SplitStream [CDK+03] löst dieses Problem, indem mehrere Bäume aufgebaut werden. Jeder innere Knoten in einem Baum ist in allen anderen Bäumen ein Blatt. Damit ist auch die Last auf die Knoten besser verteilt.

Heterogene Bandbreiten

Ein weiteres Problem tritt auf, wenn verschiedene Upload-Geschwindigkeiten der Peers berücksichtigt werden müssen. Ein einzelner Analog-Modem-Benutzer in einem inneren Knoten würde das Verteilen der Chunks im gesamten Teilbaum verlangsamen.

Das Besetzen von inneren Knoten mit mehreren Peers vermeidet dieses Problem. Ist B die Upload-Bandbreite des Seeders, werden die Knoten des Baums mit Peers besetzt, so dass die gesamte Upload-Bandbreite in einem Knoten $\geq B$ Chunks pro Runde beträgt. Die Wurzel des Baums besteht aus dem Seeder.

Der Algorithmus arbeitet nun in zwei Phasen, die den Phasen der Lower Bound für heterogene Bandbreiten (Abschnitt 4.2.1) gleichen. In Phase 1 sendet ein Knoten in einer Runde B Chunks an einen Nachfolger-Knoten. In Phase 2 werden die Chunks innerhalb des Knotens verteilt.

Um den Algorithmus mit der Lower Bound zu vergleichen, wird ein binärer Baum verwendet. Phase 1 endet, wenn jeder Knoten alle Chunks hat. Bei einem Baum der Höhe h ist dies nach $2\left[h + (C - B)\,\frac{1}{B}\right]$ Runden der Fall. Diese Runden bestehen aus $2\,(C - B)\,\frac{1}{B}$ Runden für die Verteilung aller bis auf B Chunks beim Seeder, und $2h$ Runden für die Verteilung der letzten B Chunks an die

Blätter.

Phase 2 entspricht derjenigen aus der Lower Bound und dauert in einer Gruppe mit n Peers $(n-1)\,\frac{C}{B}$ Runden. Die Anzahl der benötigten Runden für die gesamte Verteilung beträgt also:

$$\begin{aligned} & 2\left\lceil h + (C-B)\,\frac{1}{B}\right\rceil + (n-1)\,\frac{C}{B} \\ = \;& 2h + 2\frac{C}{B} - 2 + n\frac{C}{B} - \frac{C}{B} \\ = \;& 2h + \frac{C}{B}\,(n+1) - 2 \end{aligned}$$

Verwendet man Gruppen mit jeweils *gleichvielen Peers* als Knoten, enthält jeder Knoten n Peers, und der Baum folglich $\frac{N}{n}$ Knoten. Somit beträgt die Höhe des Baums $\left\lfloor \log_2 \frac{N}{n} \right\rfloor$. Die Verteilung hat also eine Gesamtdauer von

$$2\left\lfloor \log_2 \frac{N}{n} \right\rfloor + \frac{C}{B}\,(n+1) - 2$$

Runden.

Enthalten die Gruppen *Peers gleicher Bandbreite*, hat eine Gruppe mit Bandbreite B_i eine Grösse von $\frac{B}{B_i}$ Peers. Somit beträgt die Dauer

$$2h + \frac{C}{B}\left(\frac{B}{B_i} + 1\right) - 2$$

Runden für eine Gruppe mit Bandbreite B_i.

4.3.2 Hypercube

Definition (d-Dimensionaler Hypercube)**.** *Ein* d-dimensionaler Hypercube *ist ein Graph* $G = (V, E)$*, mit* $V = \{0,1\}^d$ *und* $E = \{(u_0 ... u_{d-1},\ v_0 ... v_{d-1}) \mid \sum_0^{d-1} \mid u_i - v_i \mid = 1\}$*, das heisst, zwei Knoten sind genau dann miteinander verbunden, wenn deren Hamming Distanz 1 ist. Zwei Knoten sind* Nachbarn in Dimension i*, wenn sie sich genau im* i*-ten Bit unterscheiden.*

Ein d-dimensionaler Hypercube besteht aus 2^d Knoten. Nehmen wir an, dass die Anzahl der Peers inklusive Seeder $N = 2^d$ ist, also $d = \log_2 N$. Weiter sei die Bandbreite von jedem Peer 1 Chunk pro Runde. Mit Algorithmus 1 werden C Chunks an die Peers verteilt. Dazu werden in [GS05] folgende Beobachtungen gezeigt:

- Nach $\log_2 N$ Runden hat jeder Peer genau einen Chunk.
- Nach $\log_2 N$ Runden haben die Nachbarn in Dimension $i \mod \log_2 N$ genau zwei unterschiedliche Chunks und können diese untereinander austauschen. Damit erhält ein Peer in jeder Runde genau einen Chunk.
- Ein Chunk, den der Seeder in Runde i versendet, ist in Runde $i + \log_2 N$ an jeden verteilt.

Algorithmus 1 Verteilung von C Chunks an N Peers auf Hypercube

```
for Runde i = 1 to C + log2 N do
  if seeder then
    wähle Chunk Nummer i
  else
    wähle Chunk mit der höchsten Nummer, der bisher empfangen wurde
  end if
  sende gewählten Chunk an den Nachbarn in Dimension i mod log2 N
end for
```

Abbildung 4.5 zeigt die Peers in einem 3-dimensionalen Hypercube nach $\log_2 N = 3$ Runden. Jeder besitzt genau einen Chunk und sendet diesen in der Runde 4 an seinen Nachbarn in Dimension 1 ($4 \mod \log_2 N$). Chunk 1 wurde in Runde 1 versendet und ist in Runde 4 ($1 + \log_2 N$) an jeden verteilt, was aus der Abbildung leicht ersichtlich ist. Weiter zeigt die Abbildung, dass alle Peers, ausser dem Nachbarn des Seeders, Chunks versenden.

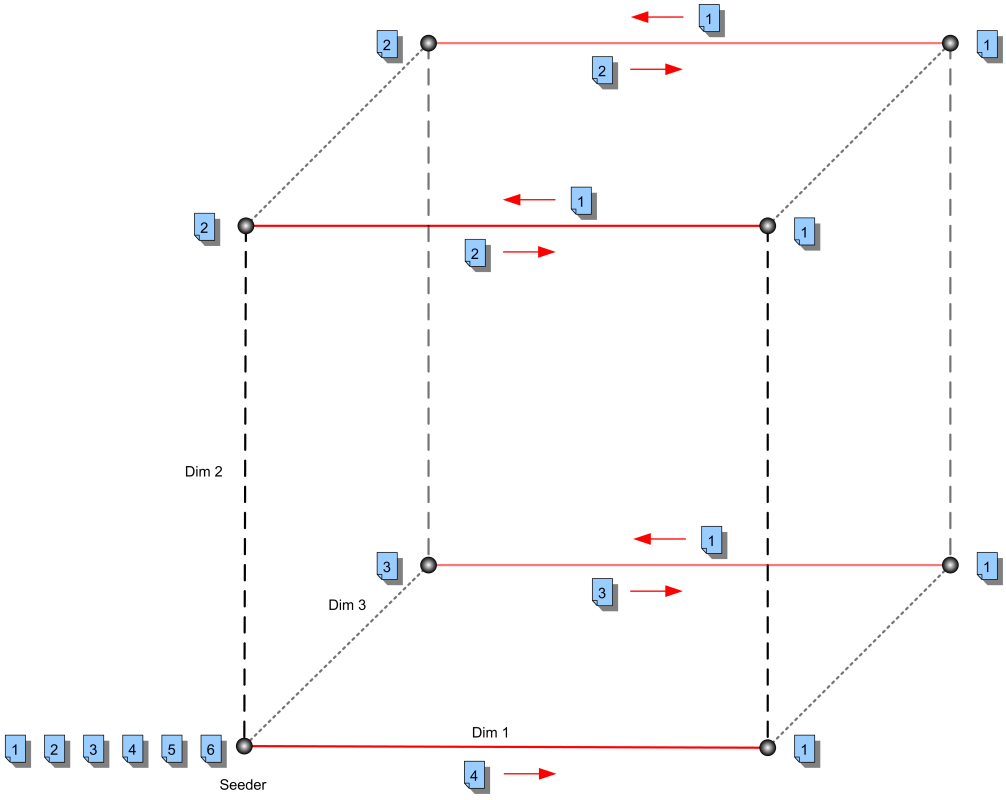


Abbildung 4.5: Verteilung auf Hypercube der Dimension 3 während Runde 4

In Runde C versendet der Seeder Chunk C, der dann in Runde $C + \log_2 N$ verteilt ist. Damit endet der Algorithmus und ist somit beinahe optimal. Das Ende des Algorithmus kann aber noch so angepasst werden ([GS05]), dass der

Seeder und andere Peers, die nach C Runden nichts mehr an ihre Nachbarn senden können, an bestimmte andere Peers senden. Damit kommt man auf optimale $\log_2 N + C - 1$ Runden.

Die erforderliche Download-Bandbreite beträgt auch 1 Chunk pro Runde, da nach $\log_2 N$ Runden jeder Peer bis auf die Nachbarn des Seeders in jeder Runde genau einen Chunk sendet und empfängt. Da in Runde i der Nachbar vom Seeder in Dimension $i \mod \log_2 N$ nichts senden kann, ist der genutzte Anteil der gesamthaft verfügbaren Upload-Bandbreite[1] $\frac{N-1}{N}$ und damit beinahe 100% für grosse N.

Weiter hat jeder Peer immer gleichviele Chunks, so dass alle Peers gleichzeitig fertig sind. Peers, die sofort nach dem Download ausfallen, sind also kein Problem. Um seltene Ausfälle während des Downloads zu berücksichtigen, können die Knoten des Hypercubes genauso wie die Knoten bei einem Baum mit mehreren Peers besetzt werden. Bei einem Ausfall eines Peers sinkt aber die Upload-Bandbreite in seinem Knoten.

Der Algorithmus wird in [GS05] auf beliebige N Peers verallgemeinert. Es wird ein Hypercube der Dimension $\lfloor \log_2 N \rfloor$ verwendet, in dem die Knoten aus entweder einem oder zwei Peers bestehen. Man kommt damit ebenfalls auf $\lceil \log_2 N \rceil + C - 1$ Runden.

Der Algorithmus funktioniert aber nur so gut, wenn alle Peers gleichzeitig ankommen. Bei einer bestimmten Ankunfts- und Ausfallrate von Peers müsste eine Möglichkeit gefunden werden, den Hypercube um eine Dimension zu vergrössern und wieder zu verkleinern. Auch ist es nicht ganz einfach, einen neuen Peer ins System einzubinden, da er erst nach $\log N$ Runden alle Chunks hat, um in jeder Runde seinem Nachbarn den richten Chunk senden zu können.

Heterogene Bandbreiten

Der aufgeführte Algorithmus kann auf unterschiedliche Upload-Bandbreiten verallgemeinert werden. Die N Peers werden wie bei den Lower Bounds für heterogene Bandbreiten in Gruppen aufgeteilt, so dass jede Gruppe eine gesamte Bandbreite von B Chunks pro Runde hat. Diese Gruppen werden nun als Knoten in einem Hypercube angeordnet. Die Knoten senden pro Runde B Chunks an ihren Nachbarn der entsprechenden Dimension.

Verwendet man Gruppen mit jeweils *gleichvielen Peers* als Knoten, dauert die Verteilung

$$\log_2 \frac{N}{n} + \frac{n}{B} C$$

Runden. Die benötigte Download-Bandbreite eines Peers i ist in Phase 1 B_i und in Phase 2 $\frac{B-B_i}{n-1}$.

Enthalten die Gruppen *Peers gleicher Bandbreite*, brauchen Peers mit Bandbreite B_i

$$\log_2 G + \frac{C}{B_i}$$

Runden. Die benötigte Download-Bandbreite dieser Peers beträgt B_i.

[1] In [QS04] wird die genutzte Bandbreite auch als *Sharing-Effektivität* bezeichnet.

Man kann aber auch mehrere Hypercubes mit Peers gleicher Upload-Bandbreite erstellen. Sei $B = B_1$ die Upload-Bandbreite des Seeders und $B_i > B_{i+1}$ mögliche Upload-Bandbreiten der Peers. Dann enthalte Hypercube 1 den Seeder und Peers mit Upload-Bandbreite $\geq B_1$. Hypercube i enthalte N_i Peers mit Bandbreiten $\geq B_i$ und $< B_{i-1}$.

Auf Hypercube 1 wird nun der Algorithmus für homogene Bandbreiten verwendet, bei dem ein Chunk pro Runde versendet wird. Nach $\log N_1$ Runden hat in Runde i der Nachbar vom Seeder in Dimension $i \mod \log_2 N_1$ einen Chunk, den er niemandem in seinem Hypercube senden kann. Dieser Chunk wird nun an den nächst-kleineren Hypercube 2 gesendet. Rekursiv bilden nun die Nachbarn des Seeders im Hypercube i einen virtuellen Seeder für Hypercube $i + 1$.

Eine Runde im Hypercube i dauert $\frac{1}{B_i}$ lange, da die Bandbreite der Peers B_i Chunks beträgt. Die Verteilung in Hypercube 1 dauert also $(C + \log_2 N_1) \frac{1}{B_1}$ lange. Nach $\log_2 N_1 \frac{1}{B_1}$ Zeit können Chunks an Hypercube 2 gesendet werden. Für Hypercube i dauert es $\sum_{j=1}^{i-1} \frac{\log_2 N_j}{B_j}$ lange, bis Peers die ersten Chunks erhalten. Die Dauer für den Download der C Chunks beträgt daher für Peers in Hypercube i:

$$\sum_{j=1}^{i-1} \frac{\log_2 N_j}{B_j} + (C + \log_2 N_i) \frac{1}{B_i} = \sum_{j=1}^{i} \frac{\log_2 N_j}{B_j} + \frac{C}{B_i}$$

Die Download-Bandbreiten für die Peers neben den virtuellen Seedern in Hypercube i betragen $B_i + B_{i-1}$, da die Runden in Hypercube i und Hypercube $i - 1$ unterschiedlich lange dauern und daher nicht synchron laufen. Also ist es möglich, dass ein Peer gleichzeitig einen Chunk aus seinem Hypercube und einen Chunk vom virtuellen Seeder empfangen muss. Alle anderen Peers benötigen eine Download-Bandbreite von B_i.

Diese Methode funktioniert optimal, wenn sich die Bandbreiten der Peers gut in Kategorien aufteilen lassen. Dies ist im Internet gegeben, da sich die Bandbreiten grob in die Kategorien LAN, Breitband und Analog-Modem unterteilen. Natürlich ist auch noch eine feinere Unterteilung möglich, aber prinzipiell lassen sich die Bandbreiten gut in konstant viele Kategorien unterteilen.

4.3.3 Zufälliger Graph

Schauen wir uns wieder das einfache Modell an, in dem sich alle Peers gleichzeitig beim Tracker anmelden. So hat der Tracker verschiedene Möglichkeiten, den zufälligen Graphen aufzubauen.

Ein zufällger Graph, in dem alle Knoten den gleichen Grad k haben, kann schrittweise aufgebaut werden. In jedem Schritt werden zwei zufällige Knoten mit einem kleinen Grad miteinander verbunden. Wenn alle Knoten Grad k haben, kann aufgehört werden. Wenn der Graph nicht zusammenhängend ist, muss nochmals von vorne begonnen werden. Der Nachteil an dieser Methode ist, dass keine Möglichkeit besteht, später Knoten hinzuzufügen, ohne den Grad oder die Nachbarschaft der bisherigen Knoten zu ändern. Die Strategie kann also nicht einfach an das komplexere Modell angepasst werden, in dem die Peers

in bestimmten Abständen beim Tracker anfragen, und dann sofort eine Antwort erwarten.

Mit dem *Peerato Tracker* wird der Graph wie in BitTorrent schrittweise mit der Ankunft der Peers aufgebaut. Zuerst besteht der Graph nur aus dem Seeder. In jedem Schritt kommt ein neuer Peer hinzu und wird vom Tracker mit k zufälligen Peers des bisherigen Graphen verbunden. Wenn Peers ausfallen, wird der Tracker wieder angefragt, um die fehlenden Peers zu ersetzen.

Zufällige Graphen sind robust gegen Ausfälle. Strukturierte Graphen sind dagegen oft aufwendiger zu unterhalten. Ausserdem vereinfachen die strukturierten Graphen Attacken, weil ein Gegner gezielt Topologieinformationen ausnutzen kann.

Jedoch können in einem zufälligen Graphen die Peers so untereinander verbunden sein, dass keine schnelle Verteilung möglich ist. Verbindet sich jeweils der neue Peer zufälligerweise immer mit den k neusten Peers, so dauert es linear lange in der Anzahl der Peers, bis ein Chunk den neusten Peer erreicht. Wie gut zufällge Graphen sind, hängt also auch davon ab, mit welcher Wahrscheinlichkeit solche pathologische Fälle eintreten.

Auswahl des nächsten Peers

Ist ein Peer erstmal mit seinen Nachbarn verbunden, kann mit dem Austausch der Chunks begonnen werden. Nach jedem gesendeten Chunk stellt sich die Frage, welchem Peer als nächstes ein Chunk gesendet wird. Natürlich könnte einfach ein *zufälliger Peer* gewählt werden. Man kann die Peers aber auch nach bestimmten Kriterien auswählen.

Einerseits sollten Peers mit *hoher Uploadrate* bevorzugt werden, da diese den Chunk danach schneller weiterverbreiten können, vorausgesetzt sie bleiben online. Ehrliche Peers können sich die Uploadrate gegenseitig mitteilen, ansonsten muss diese berechnet werden. BitTorrent verwendet dazu die lokale Downloadrate des Peers. Ein Peer optimiert seine Downloadrate momentan, indem er Chunks an Peers sendet, die ihm die beste Downloadrate liefern.

Andererseits sollte an Peers, die noch *keine Chunks* haben, möglichst schnell ein Chunk gesendet werden, damit ihre Uploadrate genutzt werden kann. Dies kann aber von einem Gegner missbraucht werden, indem er immer behauptet, er habe keine Chunks.

Eine andere Möglichkeit besteht darin, eine *Upload-Queue* in der Reihenfolge aufzubauen, in der die Peers anfragen. So muss ein Peer, der später anfragt auch länger warten. Um die Peers zum Uploaden zu motivieren, kann ein *Kredit-System* eingesetzt werden. Peers, die *mehr Chunks gesendet als empfangen* haben, werden in der Queue bevorzugt.

[GS05, BHP05, JA05] stellen einen ähnlichen Tit-for-Tat Mechanismus für BitTorrent vor. Ein Peer speichert für jeden anderen Peer die gesendeten Daten u und empfangenen Daten d. Wenn $u - d$ das *Kredit-Limit* überschreitet, werden diesem Peer keine Daten mehr gesendet. Dies verhindert aber nicht Freerider, die bei verschiedenen Peers das Kredit-Limit erreichen. Wenn genug Peers vorhanden sind, kann ein Freerider die ganze Datei downloaden, ohne uploaden zu müssen.

Auswahl des nächsten Chunks

Wenn ein Peer eine Download-Möglichkeit erhält, muss er sich für einen Chunk entscheiden, den er als nächstes erhalten will. Wenn der Peer keine Informationen über die Chunks anderer Peers hat, dann wählt er von den angebotenen Chunks einen *zufälligen Chunk*, der ihm noch fehlt.

Wenn die Peers über globales Wissen aller anderen Peers verfügen, kann die *Global Rarest First*-Strategie angewandt werden. Der Peer wählt von den Chunks des anderen den global seltensten Chunk. Meist haben die Peers jedoch nur lokales Wissen über ihre Nachbarn. Hier kann die *Local Rarest First*-Strategie verfolgt werden.

Bei der *Rarest Sent First*-Strategie wählt der *Sender* aus, welchen Chunk der Empfänger bekommt. Diese Methode scheint nicht besser zu sein als Local Rarest First, da auch der Sender keine globale Sicht hat, und ausserdem von der Nachbarschaft des Empfängers nichts weiss. Er kann auch nicht wissen, ob die Peers, denen er einen bestimmten Chunk gesendet hat, noch online sind. Ausserdem muss er dafür eine Datenstruktur verwalten, was nicht wünschenswert ist, wenn er längere Zeit online bleibt. Ein Peer, der nur zum Netz beiträgt, aber nicht davon profitiert, soll auch nicht komplizierte Berechnungen durchführen müssen und auch sonst so wenig wie möglich belastet werden.

Um die Strategien zu vergleichen, nehmen wir an, dass alle bis auf einen Chunk repliziert sind, und ein Peer nun die Möglichkeit hat, einen Chunk vom Seeder zu wählen. Sei m die Anzahl der Chunks, die dem Peer noch fehlen. Bei der Global Rarest First Strategie wird der nicht-replizierte Chunk mit Wahrscheinlichkeit 1 gewählt. Bei der Random Strategie wird von m Chunks ein bestimmter Chunk mit Wahrscheinlichkeit $\frac{1}{m}$ gewählt, dies gilt natürlich auch für den nicht-replizierten Chunk.

Bei der Local Rarest First Strategie werden die bei den Nachbarn replizierten Chunks nicht gewählt, da der Peer diese lokal sieht. Sei l die Anzahl der Chunks, die ein Peer lokal repliziert sieht, selbst aber noch nicht besitzt. Dann beträgt die Wahrscheinlichkeit $\frac{1}{m-l}$, dass der nicht-replizierte Chunk gewählt wird. Die Local Rarest First Strategie ist in diesem Fall also besser als die Random Strategie, wenn $l > 0$.

Hingegen ist die Local Rarest First Strategie schlecht, wenn ein Chunk lokal zwar als einziger nur einmal vorhanden ist, global gesehen aber viel häufiger vorkommt als die anderen Chunks. In diesem Fall wäre es besser, wenn ein anderer Chunk gewählt würde.

4.3.4 Optimierung des Chunkaustauschs

Der Tracker könnte die Peers auch so miteinander verbinden, dass der Austausch von Chunks zwischen zwei Nachbarn maximiert wird. Wenn sich der Tracker gut verteilen lässt (siehe dazu Kapitel 5), kann der Tracker auch kompliziertere Berechnungen durchführen. Zur Berechnung werden aber die Chunk-Bitfelder der einzelnen Peers gebraucht. Wenn diese nicht ständig aktualisiert werden, ist die Berechnung ungenau. Ziel ist es aber, möglichst wenig Kontakt mit dem Tracker zu haben.

Um den Tracker zu entlasten, könnten die Peers ihre Nachbarn untereinander austauschen. Ein Seeder kann den Peer zwar in die Upload-Queue eintragen, aber ihm noch einen Peer mitteilen, dem er den Chunk bereits gesendet hat. Dort kann er den Chunk vielleicht schneller erhalten.

Die Chunk-Bitfelder eignen sich für den Austausch von Peers unter Nachbarn ebenfalls gut. Es muss noch ein Mass gefunden werden, so dass der Austausch von Chunks in beide Richtungen optimiert ist.

Seien b_i das Bitfeld von Peer i, b_j das Bitfeld von Peer j. Dann ist

$$c_i = |(b_i \wedge b_j) \otimes b_i|$$

die Anzahl von Chunks, die Peer i an Peer j senden könnte.

Um einen gesamthaft maximalen Chunkaustausch zu erreichen, muss also $c_i + c_j$ maximiert werden. Damit sie sich gegenseitig etwa gleich viele Chunks geben können, muss $|c_i - c_j|$ minimiert werden.

Damit haben wir ein Mass für optimale Nachbarn gefunden:

$$c_i + c_j - |c_i - c_j|$$

Auf eine Anfrage von Peer i nach einem neuen Nachbarn sucht der Angefragte denjenigen unter seinen Nachbarn, der den obigen Ausdruck maximiert.

5 Verteilter Server

Wenn ein System mit einem zentralen Server immer funktionieren soll, muss dieser Server immer *erreichbar* sein. Er darf also nicht ausfallen und muss ganz besonders vor Attacken geschützt werden. Ausserdem kann ein Server nur eine begrenzte Anzahl von Requests behandeln. Die Datenmenge, die auf einem Server gespeichert werden kann, ist ebenfalls begrenzt.

Mit der Verteilung auf N Server wird das System *skalierbarer*. Es können dann bei homogenen Servern bis zu N mal so viele Requests behandelt werden und N mal so viele Daten gespeichert werden. Zusätzlich wird das System durch Verteilung *ausfallsicherer*. Erst wenn mehrere oder sogar alle Server gleichzeitig ausfallen, kann das System keine Requests mehr beantworten.

Um einen zentralen Server zu verteilen, müssen zuerst die Anforderungen der *Anwendung* an einen verteilten Server festgelegt werden.

- Müssen die Daten immer erreichbar sein?
- Müssen die Daten immer aktuell sein?
- Dürfen Daten verloren gehen, oder ist das kritisch?
- Wird eher die Datenmenge oder werden die Anfragen pro Sekunde zu einem Skalierungsproblem?
- Muss eine Anfrage in konstanter Zeit beantwortet werden können?

Wenn der Client von der Verteilung nichts sehen soll, kann auch ein Server verwendet werden, der nur für die Anfragen zuständig ist. Dieser leitet dann die Anfrage des Clients an den zuständigen Server weiter. Dies hilft aber nicht der Ausfallsicherheit.

Die Daten sollten redundant unter den Servern verteilt werden, so dass eine bestimmte Anzahl von Servern im gleichen Zeitintervall ausfallen kann. Durch regelmässiges Pingen aller Server kann ein Ausfall erkannt werden. Die Daten dieses Servers können dann von einem redundanten Server auf die verbleibenden verteilt werden.

In diesem Kapitel wird behandelt, welche zentralen Komponenten in SPAMATO existieren, welche Aufgaben sie haben und wie gut sie verteilt werden können. Wenn die Antwortzeit für eine Anfrage t Sekunden beträgt, können nur $\frac{1}{t}$ Anfragen pro Sekunde beantwortet werden. Steigt die Anzahl Clients, so steigt auch die Anzahl der Anfragen pro Sekunde. Übersteigt die Anzahl der Anfragen $\frac{1}{t}$ pro Sekunde, so ist man gezwungen, einen leistungstärkeren Server zu beschaffen oder den Server zu verteilen.

5.1 Tracker

Beim Tracker ist nicht die Datenmenge das Problem, sondern die Anfragen pro Sekunde. Verlorene Daten bereiten ebenfalls keine Schwierigkeiten. Ein Peer, der online ist, meldet sich in regelmässigen Zeitintervallen wieder.

5.1.1 Verteilung auf mehrere Server

Ein Tracker ist für eine maximale Anzahl von Peers verantwortlich. Ein Peer, der sich neu registrieren möchte, wählt einen Tracker aus seiner Liste aus. Hat der Tracker seine Kapazitätsgrenze erreicht, weist er neue Peers ab, gibt ihnen aber noch eine Liste der anderen Tracker. Mit dieser Liste kann er seine Tracker-Liste aktualisieren und einen anderen Tracker zur Anmeldung verwenden. Auch wenn ein Tracker nicht erreichbar ist, wird ein anderer aus der Liste gewählt.

Ein neu hinzukommender Tracker fragt einen bestehenden Tracker an und bekommt sowohl eine Liste mit allen Peers als auch eine Liste der anderen Tracker. Updates wie neue oder gelöschte Peers werden in regelmässigen Zeitintervallen unter den Trackern ausgetauscht. Jeder Tracker kann nur für diejenigen Peers Updates verschicken, für die er verantwortlich ist. Andere Peers werden lokal gelöscht, falls sich der verantwortliche Tracker zu lange nicht mehr gemeldet hat. Dies aber nur, falls sich der Peer unterdessen nicht an einen anderen Tracker gewandt hat.

Wenn es für eine Datei nur wenige Peers gibt, könnte ein Tracker auch zwischen einem Intervall andere Tracker anfragen, ob sie neue Peers zu dieser Datei haben.

5.1.2 Verteilung unter Clients

Wie schon in Abschnitt 2.2 erwähnt, können in einem DHT-Netz beliebige Daten völlig dezentral gespeichert werden. Bei einem File-Sharing System muss deshalb entschieden werden, ob die Daten selbst oder die Adressen von Peers, die diese Daten besitzen, gespeichert werden sollen.

Speichern der Chunks im DHT-Netz

In CFS [DKK+01] wird eine Datei in Chunks aufgeteilt, die in einem DHT-Netz gespeichert werden. Die Chunks sind nur wenige Kilobyte gross, womit eine grosse Datei relativ gleichmässig im DHT-Netz verteilt werden kann. In CFS sucht ein Peer zuerst nach der Datei, worauf er die Hashwerte der Chunks erhält. Danach wird nach jedem einzelnen Hashwert gesucht, so dass der Peer die Daten erhält.

Um Hotspots bei Peers mit populären Chunks zu vermeiden, werden die Chunks gecached. Jeder Peer verwaltet einen LRU-Cache worin er eine fixe Anzahl von Chunks speichern kann. Wenn ein Peer einen Chunk erhalten hat, speichert er eine Kopie des Chunks bei jedem Peer entlang des Pfades zum Uploader. Möglich wäre auch, dass die Kopie nur beim zweitletzten Peer des Pfades gespeichert wird. Damit wird praktisch der gleiche Effekt erzielt, es finden aber pro Download nur noch zwei Uploads statt.

Das System eignet sich gut für eine Gruppe von Servern, welche die Chunks speichern und nicht oft ausfallen. Es kommt aber nicht mit häufigen Ausfällen von Peers klar, da die Daten bei jedem Ausfall herumgeschoben werden müssen. Ebenfalls muss ein einzelner Peer bei temporär wenigen Peers im Netz viele Daten speichern. Ein weiterer Nachteil ist, dass Daten an Peers gesendet werden müssen, ohne dass diese die Daten verwenden.

Die Frage ist, wer die Daten ins Netz speichern darf. Ohne zentrale Instanz muss in SPAMATO jedem Peer erlaubt sein, seine Plug-Ins im Netz zu speichern. Mit der Verwendung eines Servers wäre es jedoch möglich, dass nur der Server Dateien im Netz speichern kann. Seltene Anfragen werden an den Ersteller des Plug-Ins weitergeleitet, bei steigender Popularität einer Datei wird sie im DHT-Netz verteilt. Für wenig gefragte Dateien muss so kein Aufwand für das Speichern im Netz betrieben werden. Gegner können so das Netz nicht angreifen, indem sie viele Plug-Ins ins DHT stellen.

Speichern der Adressen im DHT-Netz

Wie in Abschnitt 2.2 gezeigt, können im DHT-Netz auch Adressen von Peers gespeichert werden, die eine Datei haben. Wenn eine Datei aber sehr gesucht ist, wird der Peer, der für die Adressen der Datei verantwortlich ist, mit Anfragen überlastet. Caching ist schwierig, da sich die Menge der Adressen ständig ändert. Es befänden sich veraltete Adressen in den Caches. Da aber nicht unbedingt die Adressen aller Peers benötigt werden, könnte trotzdem eine Art Cache verwendet werden. Bei einer Registrierung oder Anfrage nach einer Datei wird folgender Algorithmus verwendet:

1. Der Anfragende sucht mit dem Key der Datei nach Adressen.
2. Ein Angefragter gibt die Adressen aller Peers zurück, die bei ihm schon mal angefragt haben. Zusätzlich sendet er die Adresse des nächsten Peers auf dem Pfad zum verantwortlichen Peer. In bestimmten Intervallen werden diese Adressen mit den nächsten Peers ausgetauscht. Die Adressen haben ein Timeout und werden gelöscht, wenn sie alt sind.
3. Wenn der Anfragende zuwenig Adressen erhalten hat, fragt er rekursiv den nächsten Peer, den er mit den Adressen erhalten hat.

Die Anzahl der Anfragen beim verantwortlichen Peer wird so verringert. Anstatt Verbindungen mit allen Peers aufbauen zu müssen, werden nur noch Verbindungen zu den Peers aufgebaut, die mit ihm Adressen austauschen.

Suche nach Plug-Ins

Ohne zentrale Instanz braucht das DHT einen Suchmechanismus für Plug-Ins. Dazu werden die Metadaten zum Plug-In verwendet. Beispielsweise könnte zu jedem Schlüsselwort aus der Beschreibung des Plug-Ins die ID des Plug-Ins gespeichert werden. Zur ID werden Informationen über das Plug-In gespeichert, wie die Version, die Beschreibung und die Hashwerte der Chunks. Bei einer Suche erhält der Benutzer eine Liste von passenden Plug-Ins mit Beschreibungen und kann eines oder mehrere Plug-Ins zum Downloaden auswählen.

Sicherheit bei Gegnern

In einem System ohne zentrale Instanz ist die Sicherheit von besonderer Bedeutung. Gegner können beliebige falsche Angaben machen, oder Informationen für sich behalten und nicht weitergeben. Es sollte daher ein DHT-System wie Kademlia [MM02] gewählt werden, das zumindest keine Routing-Attacken zulässt.

Gegner können aber auch bei der Suche falsche Angaben machen, wie zum Beispiel falsche Hashwerte der Chunks. Ein Download der Datei wird dadurch verunmöglicht. Mit einer zentralen Instanz könnte sich der Ersteller eines Plug-Ins die Hashwerte und die Beschreibung des Plug-Ins signieren lassen, so dass ein Benutzer sicher sein kann, dass die Hashwerte zur Datei korrekt sind.

Weiter muss verhindert werden, dass ein Gegner beliebig viele Daten oder sogar Viren ins Netz stellt. Ein Gegner kann Spam verteilen, indem er für beliebte Schlüsselwörter Dateien mit Beschreibungen speichert, die Spam enthalten, da diese bei einer Suche angezeigt würden. Auch kann er zu einem Virus eine falsche Beschreibung angeben, so dass ein Benutzer meint, es sei ein neuer Filter. Es ist deshalb zwingend notwendig, dass es entweder eine zentrale Instanz gibt, die den Source-Code überprüft, oder ein Trust-System, in dem Benutzer die Plug-Ins bewerten können.

Bei der Verwendung einer zentralen Instanz sendet der Ersteller den Source-Code an die zentrale Instanz. Diese überprüft den Code, und sendet dem Ersteller das kompilierte Plug-In und signierte Publish-Informationen, wie die Beschreibung und die Hashwerte des kompilierten Plug-Ins. Ein Benutzer kann so zwischen überprüften Plug-Ins und anderen unterscheiden und sich bei der Suche nur die überprüften anzeigen lassen.

Um die Plug-Ins zu bewerten, könnte ein verteiltes Trust-System wie in Abschnitt 5.4.2 beschrieben eingesetzt werden. Es gibt verschiedene Bewertungskriterien für Plug-Ins, wie Bösartigkeit oder Nützlichkeit. Für Bösartigkeit existiert eine globale Meinung, hingegen hängt die Nützlichkeit von der Sicht des Benutzers ab.

Plug-In-Bewertungen sind aber nicht so häufig wie E-Mail-Bewertungen, weshalb für die Benutzer keine zuverlässigen Trust-Werte aufgebaut werden können. Ein Gegner könnte aber mit vielen Accounts hohe Trust-Werte erreichen, indem er andere Plug-Ins bewertet. Wenn er dann einen Virus als gut bewertet, ist das für den Benutzer fataler als wenn eine E-Mail falsch eingeordnet wird.

Um Plug-Ins zu bewerten, kann auch jeder Benutzer eine Meinung schriftlich abgeben. Dies ist flexibler, da keine geeigneten Bewertungskriterien gefunden werden müssen. Andererseits kann ein Gegner so noch einfacher Spam im Netz verbreiten, selbst wenn in den Bewertungen URLs und Bilder nicht erlaubt sind. Es könnte aber ein Filter-System beim Benutzer eingerichtet werden, so dass er nur die Meinungen von bestimmten Personen sieht.

5.2 AAAS: Automatic Authorization and Authentication System

Beim AAAS [Sch04] benötigt ein Benutzer eine gültige E-Mail-Adresse, um sich einen Account zu erstellen. Wenn der Benutzer das erste Mal einen Account erstellt, generiert er ein Public-/Private-Key Paar und sendet dieses verschlüsselt an den Server.

Der Benutzer kann den gleichen Account auf mehreren Computern verwenden. Dazu sendet er ein *reregister* mit Angabe seiner E-Mail-Adresse an den Server. Dieser sendet die entsprechenden Public-/Private-Keys an die angegebene E-Mail-Adresse.

Das AAAS erschwert das Erstellen von vielen Accounts gleichzeitig. Um dem Trooth Server (siehe dazu 5.4) eine Spam-Mail zu melden, wird ein Account benötigt. Mit jeder registrierten E-Mail-Adresse kann nur eine gültige Stimme abgegeben werden.

5.2.1 Verteilung auf mehrere Server

Der AAAS-Server ist eine kritische Komponente, die nur auf vertrauenswürdige Server verteilt werden kann. Wenn ein Gegner die Private-Keys der Benutzer erhalten würde, könnte er beliebige falsche Stimmen abgeben. Der AAAS-Server wird aber nur benötigt, wenn ein neuer Spamato-Benutzer hinzukommt, oder wenn ein bisheriger Benutzer Spamato auf einem anderen Computer installiert. Im normalen Betrieb hat ein Ausfall keine Auswirkungen. Jedoch ist es kritisch, wenn Daten verloren gehen. Deshalb sollten die Daten immer sofort repliziert werden (Sicherungskopie).

Die Public-Keys eines Benutzers sollten auch bei normalem Betrieb für den Earl-Grey- und den Trooth-Server verfügbar sein. Deshalb sollten die Public-Keys auf diesen Servern repliziert sein.

5.2.2 Verteilung unter Clients

Da zur Speicherung der Private-Keys nur vertrauenswürdige Server verwendet werden dürfen, ist eine Verteilung unter Clients nicht möglich. Jedoch kann jeder Benutzer seinen Private-Key selbst wählen und den entsprechenden Public-Key bei jeder Verwendung mitsenden. Somit ist es aber möglich, dass ein einzelner Benutzer viele Public-Keys verwendet und somit viele Accounts besitzt. Auch muss der Private-Key manuell kopiert werden, wenn der Account auf mehreren Computern verwendet werden soll.

5.3 Earl Grey

Der Earl Grey Filter [Bur04] extrahiert die URLs in einer E-Mail und schickt diese Liste an den Earl Grey Server. Die *Domain einer URL* wird beim Server identifiziert, da es verschiedene *Encodings* für dieselbe Domain gibt. Danach werden alle Duplikate und Whitelist-Domains aus der Domain-Liste entfernt. Aus den übrigen Domains wird ein *Fingerprint* für die E-Mail generiert, der dann

als ID für die Mail verwendet wird. Für jede MailId wird gespeichert, wieviele Reports und Revokes gemeldet wurden. Diese Zahlen werden dem Benutzer beim Überprüfen einer E-Mail mitgeteilt. Intern muss noch die Abbildung der Domain-Liste ohne Entfernung der Whitelist-Domains auf die MailId verwaltet werden, damit bei Änderung der Whitelist die bisherigen Stimmen zur E-Mail nicht verloren gehen.

Der Earl Grey Server wird bei jeder Überprüfung einer E-Mail verwendet und wird deshalb im normalen Betrieb unerlässlich. Die Anfragen an den Server sind häufig und nehmen mit der Anzahl der Benutzer zu. Deshalb kann der Server zur Ausfallsicherheit und Skalierbarkeit verteilt werden.

5.3.1 Verteilung auf mehrere Server

Die Verteilung könnte ähnlich wie beim Tracker erfolgen, da ein beliebiger Server die Aufgabe des Earl Grey Servers übernehmen kann. Jedoch müssen die Daten im Gegensatz zum Tracker persistent sein.

Für eine Anfrage verwendet ein Client einen beliebigen Server. Wenn dieser nicht erreichbar ist, wird ein anderer Server verwendet. Dazu sind die Whitelist und die Domains mit den Fingerprints auf allen Servern repliziert. Updates können sofort an die anderen Server verteilt werden, da sie nur bei Reports oder Revokes von E-Mails vorkommen. Ein neuer Server sucht sich einen Server aus und bekommt von ihm die Whitelist und die Domains. Ein Server, der temporär ausgefallen ist, fragt nur nach Updates seit er das letzte Mal online war. Dies kann mit Timestamps gelöst werden.

5.3.2 Verteilung unter Clients

Da sich die Daten gut nach der MailId partitionieren lassen, können sie auch gut mit dem Key *MailId* in einem DHT-Netz gespeichert werden. Jedoch können nicht einfach die Anzahl der Reports und Revokes gespeichert werden, da ein Gegner dann beliebige Zahlen zurückliefern könnte. Daher werden zur MailId die Reports und Revokes der Benutzer gespeichert. Als UserId wird der Public-Key des Benutzers verwendet. Mit dem Private-Key werden dann MailId und Evaluation signiert, um falsche Stimmabgaben zu verhindern.

Dank der Signierung der Stimmen kann ein Gegner bei einer Anfrage keine falschen Stimmen zurückliefern. Ein Gegner könnte aber gewisse Stimmen nicht zurückliefern. Auch wird damit nicht verhindert, dass ein Gegner verschiedene Public-Keys verwendet und viele Stimmen abgibt. Daher ist es bei der völlig dezentralen Variante zwingend erforderlich, dass ein Trust-System für die Benutzer (Abschnitt 5.4) eingesetzt wird.

Der Fingerprint der E-Mail kann lokal generiert werden, da eine Partitionierung nach der URL für das Decoding nicht sinnvoll ist. Dazu braucht der Client stets aktuelle Software zum Decoden einer URL. Die Verwaltung der globalen Whitelist stellt dann jedoch ein Problem dar, da sie bei jedem Client vorhanden sein muss, um den Fingerprint generieren zu können. Die einzelnen Domains der Whitelist könnten jedoch in einem DHT gespeichert werden. Um die Privacy der Benutzer bei einer Anfrage zu wahren, werden Hashwerte der Domains

verwendet. Mit einer Contains-Anfrage wird dann für jede Domain einer E-Mail gefragt, ob sie in der Whitelist ist. Dies produziert jedoch verhältnismässig viele Nachrichten. Auch bei einer kleinen Whitelist muss bei jeder Domain einer E-Mail eine Anfrage gestartet werden.

5.4 Trooth

Ein kollaboratives Spam-Filter System wie Earl Grey muss mit Gegnern zurechtkommen, die falsche Stimmen abgeben, um entweder Spam ungefiltert zu lassen oder False Positives zu generieren und damit das System unbrauchbar machen. Es muss aber auch mit E-Mails zurechtkommen, bei denen unter den Benutzern keine klare Meinung herrscht, wie zum Beispiel Newslettern.

Trooth [Sch04] ist ein Trust-System, das diese Anforderungen erfüllt. Der Trooth-Server speichert dabei lediglich die Klassifizierung eines Benutzers zu einem Objekt (ObjectId, UserId, Klassifizierung). In Spamato sind das die Reports und Revokes einer E-Mail. Die Einteilung des Objekts in eine Klasse erfolgt beim Client.

Wenn die überwiegende Mehrheit die gleiche Stimme abgegeben hat (*Normal Voting*), sendet der Server dem Client die Anzahl der Reporter und Revoker. Der Client berechnet daraus, wie das Objekt klassifiziert werden soll.

Wenn sich die Benutzer nicht einig sind, wird ein *Special Voting* durchgeführt. Der Server sendet dem Client nicht nur die Anzahl sondern eine Menge von Reportern und Revokern. Daraus wählt der Client die k Voter, die den höchsten Trust-Wert haben, und berechnet daraus, ob die E-Mail Spam ist oder nicht. Der Server könnte dem Client auch alle Reporter und Revoker schicken, jedoch ist das ineffizient bei vielen Votern. Daher bildet der Server die IDs der Reporter und Rekoker zusammen mit dem anfragenden Client auf zwei Ringe ab. Dann werden diejenigen Abschnitte der Ringe gesendet, die um den Client liegen[1].

Bei einem Report oder Revoke des Clients werden dem Client ebenfalls die Abschnitte der Ringe geschickt. Dann sieht der Client, wer so wie er selbst abgestimmt hat, und wer anderer Meinung ist. Die lokalen Trust-Werte werden dann entsprechend angepasst.

5.4.1 Verteilung auf mehrere Server

Die Daten, die auf dem Trooth-Server gespeichert werden, können nach der ObjectId horizontal partitioniert werden. Jeder Server weiss dabei stets, welcher Trooth-Server für welche ObjectId verantwortlich ist. Ein Client kann so einen beliebigen Server anfragen, der ihn an den zuständigen Server weiterleitet. Um eine Anfrage in nur einem Hop beantworten zu können, ohne dass der Client wissen muss, welcher Server für welche ObjectId verantwortlich ist, können die Daten natürlich auch vollständig repliziert werden.

[1] Es wird angenommen, dass die gleiche Gruppe von Benutzern etwa die gleichen E-Mails erhalten. Daher wird der Abschnitt meist die gleichen Voter enthalten, so dass die Menge der lokalen Trust-Werte beschränkt bleibt.

5.4.2 Verteilung unter Clients

Bei Trooth werden im Prinzip dieselben Daten gespeichert werden wie bei der Verteilung des Earl Grey Servers unter Clients. Mit der *ObjectId* als Key werden der Public-Key und die Evaluation der Benutzer signiert im DHT-Netz gespeichert.

Ein Nachteil dieser dezentralen Variante ist, dass es auch bei einem *Normal Voting* nicht genügt, lediglich die Anzahl der Reporter und Revokter zu senden. Ein Client kann nicht darauf vertrauen, dass die Signatur der Stimme richtig überprüft wurde, oder dass die Reporter und Revoker richtig gezählt wurden. Deshalb müssen auch bei einem *Normal Voting* Abschnitte des Rings oder sogar der ganze Ring gesendet werden. Dann kann der Client die Signaturen selbst überprüfen und die Stimmen auszählen.

5.5 Statistik

Beim Statistik-Server sind die Anforderungen an die Verteilung geringer. Die Daten müssen nicht aktuell sein, sie könnten auch einen Tag alt sein, oder einfach einen Trend anzeigen (Fuzzy-Statistik). Ausserdem muss man nicht jederzeit auf die Statistiken zugreifen können. Es dürfen aber keine Daten verloren gehen. Eine Statistik, bei der zu gewissen Zeiten keine Daten vorhanden sind, ist nicht aussagekräftig.

5.6 Spamato Webseite

Der Web-Server wird im normalen Betrieb nicht benötigt. Es wäre aber trotzdem wünschenswert, wenn er immer erreichbar ist. Ein neuer Benutzer sollte jederzeit Informationen holen können, und nicht durch einen Server, der offline ist, abgeschreckt werden. Dies kann mit herkömmlichen Methoden zur Verteilung von Web-Servern erreicht werden, wie zum Beispiel Replikation in Verbindung mit einem Round-Robin System. Für Downloads könnte ein Bootstrap Installer verwendet werden, der lediglich Peerato beinhaltet. Die restlichen Plug-Ins werden von den anderen Peers bereitgestellt.

6 Zusammenfassung

Spamato ist ein kollaboratives, erweiterbares Spam-Filter System. Beinahe alle Komponenten in Spamato sind Plug-Ins und können dynamisch ausgetauscht werden. Ziel dieser Arbeit war es, ein verteiltes Plug-In System zu erstellen, in dem Entwickler ihre eigenen Plug-Ins anderen Benutzern anbieten können. Zudem sollten zentrale Server-Komponenten von Spamato verteilt werden, um das System skalierbarer zu gestalten und eine höhere Ausfallsicherheit zu gewährleisten.

Peerato ist ein Peer-to-Peer System zur Verteilung von Dateien. Durch die dezentrale Funktionsweise ist das System skalierbar und ermöglicht es vielen Benutzern gleichzeitig, eine Datei zu downloaden. Dazu wurde ein Protokoll entwickelt, das es den Peers erlaubt, die Dateien untereinander auszutauschen. Die Peers lernen mit Hilfe des Trackers andere Peers kennen, die an derselben Datei interessiert sind. Wie andere File-Sharing Systeme unterteilt Peerato die Datei in Chunks, um die Parallelität zu erhöhen. Die bereits erhaltenen Chunks sind für andere Peers schon verfügbar bevor der Peer die komplette Datei hat.

In einem nächsten Schritt wurde Peerato in das Plug-In System von Spamato integriert. Dazu wurde der Update-Mechanismus erweitert, so dass das Suchen und Downloaden von Plug-Ins nicht mehr über denselben Server erfolgen muss. Der Such-Server kann wie bisher ein Web-Server sein und stellt eine Liste von Plug-Ins mit Beschreibung und Download-URL zur Verfügung. Die Download-URL kann ein beliebiges Protokoll verwenden, darunter auch das Peerato-Protokoll. Damit wird dem Plug-In System gesagt, dass der Download mit Peerato von anderen Peers erfolgen soll.

Das Publizieren von Plug-Ins hat sich dank Peerato stark vereinfacht und automatisiert. Der Entwickler wählt über die WebConfig-Seite von Spamato die Zip-Datei des Plug-Ins, eine Download-URL und einen Publish-Server. Die Download-URL und weitere, automatisch extrahierte Informationen über das Plug-In werden dann dem Publish-Server geschickt. Verwendet die Download-URL das Peerato-Protokoll, wird die Datei zusätzlich mit Peerato den anderen Peers angeboten.

Weitere Schwerpunkte dieser Arbeit bilden theoretische Überlegungen und Berechnungen zur effizienten Verteilung einer Datei. Die Effizienz der Verteilung hängt davon ab, wie die Peers miteinander verbunden sind. Zusätzlich muss ein Peer wissen, welchen Chunk er als nächstes welchem Peer schickt. Strukturierte Topologien wie Baum und Hypercube lassen Vergleiche mit der Lower Bound zu, eignen sich aber nicht für Umgebungen, in denen jederzeit ein Peer ausfallen oder hinzukommen kann. Zufällige Verbindungen zwischen den Peers lassen keine exakten Analysen zu, haben sich aber in Umgebungen mit häufigen Ausfällen bewährt.

Neben dem Download-Server für Plug-Ins gab es in Spamato noch andere

zentrale Komponenten. Für diese wurden die Aufgaben festgestellt und Anforderungen an eine verteilte Variante bestimmt. Daraus liessen sich Ideen zur Verteilung ableiten und deren Vor- und Nachteile diskutieren.

7 Ausblick

7.1 Unterstützung von Firewalls

PEERATO unterstützt derzeit nur Peers, die für andere Peers unter einem bestimmten Port erreichbar sind. Ist der Port bei einem Peer von einer Firewall oder einem Router blockiert, kann er zwar Verbindungen zu anderen Peers aufbauen, selbst aber nicht durch andere kontaktiert werden. Das Downloaden wird so erschwert, da die Verbindung nach einem Download-Request geschlossen wird, wenn beim anderen Peer kein freier Upload-Slot vorhanden ist.

Um Peers hinter einer Firewall zu unterstützen, braucht es eine Methode zum Erkennen dieser Peers. Der Tracker kann dies feststellen, indem er einen anderen Tracker ein Ping an den Peer senden lässt. Wenn er zu lange keine Antwort erhält, teilt er dem Peer mit, dass er eine Firewall hat. In diesem Fall sendet ihm der Tracker einen Buddy[1]. Mit diesem Buddy bleibt die Verbindung ständig bestehen. Ein anderer Peer kann eine Verbindung zu diesem Peer aufbauen, indem er dem Buddy einen Aufbaurequest sendet. Dieser leitet den Request an den Peer weiter, worauf dieser eine Verbindung zum Requester aufbaut.

Zwei Peers, die beide eine Firewall haben, können nicht miteinander kommunizieren, da keiner eine Verbindung zum anderen aufbauen kann, um Daten zu senden. Die Daten könnten zwar über einen Buddy gesendet werden, dieser wäre dann aber überlastet, besonders wenn er mehr als einen Firewall-Benutzer verwaltet. Alle Daten müssen zweimal gesendet werden, einmal vom sendenden Peer und einmal vom Buddy.

Weitere Probleme können mit Routern auftreten. Ein Router ermöglicht Port-Umwandlungen (Port-Triggering), so dass der Port, der von aussen sichtbar ist, an einen anderen Port im Netz weitergeleitet wird. Ein solcher Peer müsste also einen anderen Port an den Tracker senden, als den, mit dem er den Server-Socket für andere Peers erstellt.

7.2 Neighbor Selection Strategien

PEERATO erlaubt das Austauschen der Strategien, um die spezielle Eigenschaften einer Anwendung ausnützen zu können. Bisher sind aber nur wenige Strategien implementiert. Beim Tracker gibt es den *RandomSelector*, der die *NeighborSelectionStrategy* implementiert. Beim Peerato Plug-In wird der nächste Chunk mit der *RarestChunkFirst* Strategie gewählt. Wird ein Upload-Slot frei, wird mit der *FirstInQueue* oder der *CreditQueue* der nächste Peer gewählt.

[1] Ein Buddy ist ein Peer ohne Firewall, der bereit ist, für Peers mit Firewall als Proxy zu agieren

Diese Strategien sind für SPAMATO gut geeignet. In anderen Anwendungen, wie zum Beispiel Streaming, können andere Strategien erforderlich sein. Bei Streaming bildet die *NeighborSelectionStrategy* einen Verteil-Baum. Die Wurzel des Baums ist der Seeder. Innere Knoten sind Peers mit hoher Uploadrate, da sie doppelt so viel uploaden müssen wie downloaden. Analog-Modem-Benutzer bilden die Blätter des Baumes. Die *PeerSelectionStrategy* sendet abwechslungsweise an die beiden Kinder. Für Streaming wird eine *ChunkSelectionStrategy* benötigt, welche die Chunks in der richtigen Reihenfolge anfordert.

7.3 Ausbau der Sicherheit

Mit dieser Arbeit wird es jedem Entwickler ermöglicht, seine eigenen Plug-Ins zu veröffentlichen. Dies hat aber auch Nachteile, da ein Benutzer nun nicht mehr zwischen sicheren und unsicheren Plug-Ins unterscheiden kann. Deshalb wurden die Publish-Server in sichere und unsichere Server unterteilt. Auf einem sicheren Server können nur bestimmte Benutzer Plug-Ins veröffentlichen.

Denkbar ist aber auch ein System, bei dem ein Entwickler sein Plug-In signieren kann. Dann kann dem Benutzer eine Warnmeldung angezeigt werden, wenn er ein nicht signiertes Plug-In installiert, oder wenn ein signiertes Plug-In von einem Plug-In überschrieben werden soll, dessen Ersteller nicht mit dem des installierten Plug-Ins übereinstimmt.

7.4 Verteilter Server

Kapitel 5 beinhaltet einige Ideen, wie die zentralen Komponenten von SPAMATO verteilt werden können. Werden diese ausgearbeitet und implementiert, gefährdet der Ausfall eines Servers nicht den Betrieb. Ausserdem können so mehr Benutzer bedient werden.

Literaturverzeichnis

[ABW05] Keno Albrecht, Nicolas Burri, and Roger Wattenhofer. Spamato - An Extendable Spam Filter System. In *2nd Conference on Email and Anti-Spam (CEAS)*, July 2005.

[BHP05] Ashwin R. Bharambe, Cormac Herley, and Venkata N. Padmanabhan. Analyzing and Improving BitTorrent Performance. Technical Report MSR-TR-2005-03, Microsoft Research, February 2005.

[Bur04] Nicolas Burri. Spamato: A Collaborative Spam Filter System. Diplomarbeit, Swiss Federal Institute of Technology Zurich, 2004.

[CDK+03] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: High-bandwidth multicast in cooperative environments. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[Coh03] Bram Cohen. Incentives Build Robustness in BitTorrent. In *1st Workshop on the Economics of Peer-to-Peer Systems*, June 2003. `http://www.bittorrent.com`.

[DKK+01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *18th ACM Symposium on Operating Systems Principles (SOSP)*, Chateau Lake Louise, Banff, Canada, October 2001.

[ed2] eDonkey. `http://www.edonkey2000.com/`.

[GBL+03] Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robbert van Renesse. Kelips: Building an Efficient and Stable P2P DHT Through Increased Memory and Background Overhead. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[GS05] Prasanna Ganesan and Mukund Seshadri. On Cooperative Content Distribution and the Price of Barter. In *25th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2005.

[Hit02] Ron Hitchens. *Java NIO*. O'Reilly, first edition, August 2002.

[JA05] Seung Jun and Mustaque Ahamad. Incentives in BitTorrent Induce Free Riding. In *ACM SIGCOMM Workshop on Economics of Peer-to-Peer Systems*, August 2005.

[Mei05] Remo Meier. Spamato Plug-In Architecture. Semesterarbeit, Swiss Federal Institute of Technology Zurich, 2005.

[MM02] Petar Maymounkov and David Mazieres. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *1st International Workshop on Peer-to-Peer Systems (IPTPS)*, March 2002. `http://kademlia.scs.cs.nyu.edu`.

[NBRA04] J.R. Nicoll, M. Bateman, A. Ruddle, and C. Allison. Challenges in Measurement and Analysis of the BitTorrent Content Distribution Model. In *International Postgraduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting*, Liverpool John Moores University, 2004.

[PGES05] J.A. Pouwelse, P. Garbacki, D.H.J. Epema, and H.J. Sips. The BitTorrent P2P File-Sharing System: Measurements and Analysis. In *4th International Workshop on Peer-to-Peer Systems (IPTPS)*, February 2005.

[QS04] Dongyu Qiu and R. Srikant. Modeling and Performance Analysis of BitTorrent-Like Peer-to-Peer Networks. In *ACM SIGCOMM Conference*, August 2004.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, November 2001.

[Sch04] Simon Schlachter. Spamato Reloaded. Masterarbeit, Swiss Federal Institute of Technology Zurich, 2004.

[SMK+01] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM Conference*, August 2001.