INSTITUT FÜR INFORMATIK
Datenbanken und Informationssysteme

Universitätsstr. 1 D–40225 Düsseldorf

HEINRICH HEINE
UNIVERSITÄT
DÜSSELDORF

Entwicklung und Evaluierung einer Java-Schnittstelle zur Clusteranalyse von Peer-to-Peer-Netzwerken

Benjamin Treeck

Bachelorarbeit

Beginn der Arbeit:	13. Mai 2005
Abgabe der Arbeit:	27. Juli 2005
Gutachter:	Prof. Dr. Stefan Conrad
	Prof. Dr. Martin Mauve

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 27. Juli 2005 Benjamin Treeck

Inhaltsverzeichnis

1 Motivation

Tauschbörsen über das Internet zum Herunterladen und Anbieten von Dateien erfreuen sich zunehmend größerer Beliebtheit. Diese so genannten Peer-to-Peer-Netzwerke machten laut einer Studie von CacheLogic im Juni 2004 in Europa ca. 55% des gesamten Traffics im Internet aus, in Asien sogar über 80% [Par04]. Es ist daher Ziel dieser Arbeit, ein großes P2P-Netzwerk anhand Methoden der „Knowledge Discovery in Databases“ (KDD) [ES00] zu analysieren. Das auf diese Weise extrahierte Wissen soll dazu dienen, die Struktur des Netzwerks zu durchleuchten.

Unter anderem wird es Aufschluss über eventuelle Muster in der Anzahl und Größe bestehender Server geben, indem ähnliche Server im Rahmen der Clusteranalyse zu Gruppen zusammengefasst werden. Weiterhin wird untersucht werden, inwieweit Gruppen von Clients auffindbar sind, die ähnliche Dateien anbieten, also quasi Interessenkreise herausfiltern. Dazu sollen möglichst viele Clients erreicht werden, um diese dann ebenfalls einem geeigneten Clusteringalgorithmus im Rahmen des Data Minings [ES00] zu unterziehen, wobei hohe Anteile an gleichen Dateien zu gleichen Clustern führen. Die resultierenden Muster werden dann evaluiert, z.B. auf die Anzahl einbezogener Clients hin. Ein weiterer Punkt wird die Qualität der Clusterings sein, d.h. ob die gefundene Struktur sich als brauchbar herausstellt und von welchen Faktoren dies abhängt. Um diesen Fragen nachgehen zu können, wurde im Rahmen dieser Arbeit eine Schnittstelle zu einem P2P-Netzwerk in der Programmiersprache Java entwickelt.

Im Folgenden soll nach einer kurzen Übersicht über die aktuell verbreiteten P2P-Architekturen speziell auf die Kommunikation im eDonkey-Netzwerk eingegangen werden, da diese das Grundgerüst des entwickelten Programms darstellt. Es folgt eine Beschreibung des Programmkerns. Im Hauptteil sollen dann die Ideen und Implementierungen des Server Clusterings sowie die des Client Clusterings mit abschließender Bewertung der Ergebnisse aufgeführt werden.

2 P2P-Netzwerke

2.1 Übersicht

Unter Peer-to-Peer versteht man die Kommunikation unter gleichberechtigten Hosts. Es haben sich vier große P2P-Netzwerke, basierend auf diesem Prinzip mit dem Ziel eines dezentralen Datenaustauschs, in den letzten Jahren herausgebildet: BitTorrent, eDonkey/eMule, FastTrack (Kazaa, Morpheus) und Gnutella [Par04]. Dabei ist zwischen drei Architekturen zu unterschieden: dem „Centralized Directory“, dem Modell des "Query Floodings“ sowie einer Mischform der beiden, bei der gewöhnliche Peers die Rolle von Servern übernehmen. [KR05].

2.1.1 Centralized Directory

Bei dem Modell des Centralized Directory pflegen Server ein umfangreiches, zentralisiertes Verzeichnis der Dateien ihrer Clients [KR05]. Wenn diese sich zum Server verbinden, übermitteln sie dabei ihre IP-Adresse sowie eine Liste von Dateien, die sie aktuell (vollständig oder meistens auch partiell) besitzen. Für die Dateisuche befragen sie dann den Server, welcher in seinem Verzeichnis die zugehörigen Client-IP-Adressen nachschlägt. Er sendet dem anfragenden Client die Adressen von einem bzw. mehreren Zielclients, woraufhin der Anfrager sich mit diesen für einen dezentralen Dateitransfer verbindet. Ein Nachteil dieser Architektur ist, dass die Existenz des gesamten Netzwerks abhängig vom Bestand der Server ist („Single Point of Failure“). Außerdem sind diese bei tausenden von Anfragen in der Sekunde enormem Traffic ausgesetzt. Das im weiteren Verlauf betrachtete eDonkey-Netzwerk, aber auch das bekannte BitTorrent-Netzwerk, basieren auf dieser Architektur.

2.1.2 Query Flooding

Eine völlig dezentrale Variante ist Query Flooding [KR05] (z.B. Gnutella). In diesen Netzwerken besteht keine Hierarchie mehr. Jeder Knoten ist gleichwertig. Anfragen werden über Nachbarknoten weitergeleitet bis ein Knoten die Kriterien erfüllt. Dieser kontaktiert dann den anfragenden Peer. Niemand pflegt Inhaltsverzeichnisse außer seinem eigenen. Nachteilig zeigen sich jedoch u.a. die Reichweiten der Anfragen, da diese durch eine feste maximale Anzahl Hops begrenzt sind. Außerdem ist ein höheres Verkehrsaufkommen zu verzeichnen, da Anfragen nicht zu einem zentralen Server, sondern zu verschiedenen Peers weitergeleitet werden.

2.1.3 Halbzentralisiertes Modell

Ein dritter Ansatz greift Ideen von beiden oben behandelten Modellen auf und weist eine nur geringe Hierarchie auf [KR05] (z.B. FastTrack). Es werden dabei gewöhnliche Peers mit hoher Bandbreite zu Gruppenleitern bestimmt, welche die Inhaltsverzeichnisse all ihrer Gruppenmitglieder kennen. Außerdem kommunizieren sie mit anderen Gruppenleitern um Dateianfragen weiterzuleiten. Dieses Modell weist die Probleme beider vorhergehenden Ansätze auf, allerdings nur noch in geringem Ausmaß.

2.2 Protokolle

Jedes P2P-Netzwerk unterliegt einer eigenen Spezifikation von vordefinierten Nachrichten, mit denen die teilnehmenden Hosts miteinander kommunizieren. Diese Nachrichten bestehen aus Bytesequenzen, wobei für je-

de Nachricht genau festlegt ist, welche Bytes welche Informationen ausdrücken sollen.

Zuerst wird meist ein Header gesendet, in dem z.B. der Typ der Nachricht und die Länge des Payloads zu finden sind. Die Bytes des Payloads sind dann nachrichtenspezifisch zu interpretieren. Das Gnutella-Protokoll z.B. kommt mit fünf solcher Nachrichten aus [KM02]. Das eDonkey-Protokoll hingegen besteht aus ca. 65 Nachrichten [Kli04] und ist somit um einiges komplizierter. Das inzwischen noch bekanntere P2P-Netzwerk eMule basiert auf einem Protokoll, das eine Erweiterung des eDonkey-Protokolls um ca. elf Nachrichten darstellt. Es hat sich daher als Herausforderung herausgestellt, eine Schnittstelle zu entwickeln und zu evaluieren, die auf jenem komplexen eDonkey/eMule-Protokoll basiert.

Obwohl der in C++ geschriebene Code des offiziellen eMule-Clients (http://www.emule-project.net) frei zugänglich ist, wird von offizieller Seite keine Protokollspezifikation angeboten. Aus diesem Grund musste bei der Entwicklung zum einen auf inoffizielle Spezifikationen zurückgegriffen werden, zum anderen wurde auch anhand des offiziellen eMule-Clients (Version 0.45b) versucht, in Verbindung mit Ethereal, einem Programm zur Netzwerkprotokoll-Analyse (http://www.ethereal.com), unbekannte Bytefolgen zu identifizieren.

3 Das eDonkey-Protokoll

3.1 Eine zentralisierte Architektur

Im eDonkey/eMule-Netzwerk, welches auf dem Modell des Centralized Directory basiert, ist jeder eDonkey/eMule-Client in der Regel mit einer Liste von Server-IP-Adressen vorkonfiguriert [KB05]. Wenn er dann dem Netzwerk beitreten will, initiiert er eine TCP-Verbindung zu einem dieser Server. Von nun an ist es ihm (wie in 2.1.1 beschrieben) möglich, Informationen über gesuchte Dateien vom Server abzufragen, wie z.B. IP-Adressen von Clients, die diese Datei anbieten. Daraufhin wird der Client versuchen, viele simultane TCP-Verbindungen zu diesen Clients herzustellen und sich für die jeweilige Datei in eine Warteschlange einordnen zu lassen. Es steht ihm frei, sich bei verschiedenen Clients für dieselbe Datei anzumelden, mit dem Ziel, gleichzeitig verschiedene Fragmente der Datei herunterzuladen. Währenddessen werden in gleicher Weise TCP-Verbindungen von anderen Clients akzeptiert und Fragmente der eigenen Dateien hochgeladen. Das Protokoll bietet zudem einige optionale Nachrichten über das verbindungslose Transportprotokoll UDP an [Kli04].

3.2 Client-Server-Verbindung

Einen wesentlichen Bestandteil der Kommunikation im Netzwerk und somit auch der Implementierung stellt der Handshake zwischen Client und Server dar, welcher im Folgenden erläutert werden soll.

Sobald die TCP-Verbindung zum Server etabliert ist, sendet jeder Client eine „Login“-Nachricht an den Server [KB05]. Grundsätzlich besitzen eDonkey/eMule-Nachrichten immer einen Header, der aus den drei Feldern Protokoll (0xe3 für eDonkey, 0xc5 für eMule und 0xd4 für eMule komprimiert), Größe (der Nachricht in Bytes außer dem Header) und Typ (der Nachricht) besteht. Die „Login“-Nachricht enthält zudem lokale Informationen wie den TCP-Port (Standard: 4662), den UDP-Port (Standard: 4672), den Benutzernamen und eine zum größten Teil zufallsgenerierten 16 Byte großen User ID. Die User ID identifiziert eine Sitzung des Clients und ist hauptsächlich für das Credit-System relevant, bei dem ein Client für viel Upload bei anderen Clients belohnt wird, worauf hier aber nicht näher eingegangen werden soll.

Nachdem der Server nun die Anfrage erhalten hat, wird er zunächst überprüfen, ob der vom Client angegebene TCP-Port erreichbar ist oder ob der Client sich z.B. hinter einer Firewall befindet, denn dann wäre er für andere Clients nicht zugänglich [KB05]. Dazu übernimmt der Server die Rolle eines Clients und versucht eine zweite TCP-Verbindung zum angegeben Port des Clients herzustellen. Gelingt diese Verbindung und ist der anschließende Client-Client-Handshake (siehe 3.3) erfolgreich, beendet der Server die Rückruf-Verbindung wieder und weist dem Client über die initiale Verbindung anhand einer „ID Change“-Nachricht eine „hohe“ Client-ID zu. Eine solche ID leitet sich direkt von der IP-Adresse des Clients ab [Kli04]. Das heißt, so lange sich die IP nicht ändert, wird er von allen Servern die gleiche ID zugewiesen bekommen. Hat die ID einen hexadezimalen Wert von 0xAABBCCDD („Little Endian“-Repräsentation), so ist die IP des Clients DD.CC.BB.AA. Kann der Server den Port des Clients nicht erreichen, weist er ihm daraufhin eine „niedrige“ ID zu. Diese entspricht nicht mehr der IP-Adresse, sondern hat höchstens den Wert 0x00FFFFFF. Möchte ein Client mit hoher ID einen Client mit niedriger ID erreichen, muss zuerst der Server instruiert werden, den anderen Client um Rückruf zu bitten [KB05]. Daher ist es auch nicht möglich, dass zwei Peers mit niedriger ID miteinander kommunizieren, denn die Ports beider Clients sind nicht erreichbar und beide können nicht zurückgerufen werden.

In drei Fällen kann es jedoch passieren, dass der Server den Verbindungswunsch des Clients gänzlich ablehnt:

- Die IP des Clients wird blockiert („blacklisted“) aufgrund zu aggressiven Clientverhaltens.
- Der Server ist voll, hat also die maximale Anzahl an User, die er be-

dienen kann, erreicht („Hard Limit“).

- Der Port des Clients ist nicht erreichbar und das „Soft Limit“ des Servers ist erreicht, denn wenn schon eine gewisse Anzahl Clients mit dem Server verbunden ist, nimmt er nur noch Verbindungswünsche von Clients mit hoher ID entgegen, um nicht zu großer Last ausgesetzt zu sein.

Er sendet dann anstatt der „ID Change“-Nachricht eine entsprechende Fehlermeldung und trennt danach die Verbindung.

Sind Verbindungsaufbau und Handshake jedoch erfolgreich gewesen und wurde die „ID Change“-Nachricht übermittelt, sendet der Server nun meistens noch eine Reihe von „Server Message“-Nachrichten. Die Payloads dieser Nachrichten sind als Strings zu interpretieren, die entweder als Begrüßung, Fehlermeldung, Serverbeschreibung oder auch als Werbungsträger dienen.

Die Verbindung zwischen Client und Server bleibt nun über die ganze Sitzung bestehen. Der Server sendet alle vier Minuten eine „Server Status“-Nachricht an den Client um ihn über die Anzahl verbundener Benutzer und bekannter Dateien zu informieren. Der Client kann von Zeit zu Zeit diverse Dienste des Servers in Anspruch nehmen, wie z.B. die Dateisuche oder die Quellensuche für eine bestimmte Datei. Ein Client hat zudem die Möglichkeit, andere Server aus seiner lokalen Serverliste via UDP nach Dateien oder Quellen zu befragen [Kli04]. Das führt dazu, dass ein Client bei der Quellensuche nicht nur auf die Clients angewiesen ist, die zum selben Server wie er verbunden sind. Ansonsten wird UDP in der Client-Server-Kommunikation noch dafür verwendet, den Status anderer Server von der lokalen Serverliste aktuell zu halten, indem diese periodisch abgefragt werden. Für unseren Client ist die UDP-Kommunikation zwischen Client und Server jedoch nicht weiter relevant.

3.3 Client-Client-Verbindung

Clients kontakieren andere Clients in der Regel immer dann, wenn sie Dateien von ihnen herunterladen möchten. Wir benötigen eine Verbindung zu einem Client jedoch nur, wenn wir eine Liste seiner Dateien einsehen wollen. Bei der Implementierung galt daher Folgendes zu beachten:

Hat ein Peer sich zu einem anderen Peer verbunden, sendet er, so sieht es das eDonkey-Protokoll vor, eine „Hello“-Nachricht, um allgemeine lokale Informationen wie Ports, Benutzername, Clientversion und die Adresse des Servers, zu dem er verbunden ist, zu übermitteln [KB05]. Der zweite Peer antwortet mit einer „Hello Answer“-Nachricht, welche die gleichen Informationen (auf ihn bezogen) enthält.

Die Erweiterung durch das eMule-Protokoll sieht das Übermitteln einer „eMule Info“-Nachricht unmittelbar vor dem Versenden der „Hello

Answer“-Nachricht vor. In dieser optionalen Nachricht befinden sich Informationen über den eMule-Client wie z.B. ein UDP-Versionstag, der angibt, welche UDP-Nachrichten unterstützt werden. Sie informiert auch darüber, ob Quellenaustausch unter Clients, Kommentare zu Dateien oder Kompression unterstützt werden. Der Peer, von dem die Verbindung ausgeht, antwortet mit einer fast identisch aufgebauten „eMule Info Answer“-Nachricht, sofern auch er einen eMule-basierten Client verwendet.

Das eMule-Protokoll bietet außerdem eine sichere Benutzeridentifikation zur Vermeidung von Missbrauch des Credit-Systems an [KB05]. Zur Verschlüsselung wird dabei das Kryptosystem RSA verwendet. Weiterhin ist es möglich, die Vorschau von einer Datei (bei unterstützten Dateiformaten) anzufordern.

Mit all diesen lokalen Client-Informationen ließe sich Ähnlichkeit zwischen Clients definieren. Wir beschränken uns beim Ähnlichkeitsbegriff jedoch auf eine Funktion, die es ermöglicht, die Liste der Dateien eines anderen Peers einzusehen, sofern dieser seinen Client entsprechend konfiguriert hat.

4 Grundlegendes zum Programm

4.1 Struktur

Das erstellte Programm gliedert sich in vier Pakete: Programmkern, Nachrichten, Clustering und Extraktionsroutinen. Alle Klassen, die für eine „normale“ Verbindung zum eDonkey-Netzwerk benötigt werden, befinden sich im Programmkern. Sämtliche Klassen des Nachrichten-Pakets repräsentieren die Nachrichten, die zwischen Server und Client oder Client und Client ausgetauscht werden. Davon wurden 15 implementiert, da diese für unsere Zwecke unerlässlich sind. Hinter Clustering verbergen sich Klassen, die ein solches (Centroid- oder hierarchisches Clustering) repräsentieren und diverse, nützliche Methoden zur Durchführung und Analyse des Clusterings anbieten. Die Extraktionsroutinen beinhalten die Routinen, die das eDonkey/eMule-Netzwerk nach bestimmten Informationen durchsuchen, die erhobenen Daten zum Zwecke des Data Minings einem Clusteringalgorithmus übergeben und schließlich die Ergebnisse ausgeben.

4.2 Ablauf einer einfachen Verbindung

Zu Beginn des Programms wird eine Extraktionsroutine (Client oder Server Clusterer) aufgerufen. In beiden Fällen wird zunächst zu einem ersten Server verbunden, dessen IP-Adresse und Port vom Benutzer angegeben werden müssen. Dazu wird die Klasse Connection instanziert. Diese bereitet zunächst für den Server-Login notwendige Informationen vor, wie z.B.

das Parsen von lokalen Informationen in ein Byte-Array oder das Generieren der lokalen User ID.

Sobald sie eine Verbindung zum Server hergestellt hat, sendet sie diesem eine „Login“-Nachricht. Direkt im Anschluss wird der „Client Listener“-Thread gestartet, welcher einen Server Socket auf einem vom Benutzer vorgegebenen Port erstellt und ab sofort auf eingehende Verbindungswünsche von anderen eDonkey/eMule-Peers wartet, um das resultierende Socket-Objekt dem „Client Message Processor“-Thread zu übergeben. Dieser wartet nun auf eingehende Nachrichten des Peers und reagiert auf sie entsprechend. Für gewöhnlich wird der erste Client der Server selbst sein, um zu überprüfen, ob der lokale TCP-Port erreichbar ist oder nicht (siehe 3.2).

Der „Client Message Processor“-Thread arbeitet wie folgt: Zuerst wird das Protokoll der Nachricht identifiziert. Handelt es sich dabei um eine komprimierte eMule-Nachricht (0xd4), wird sie zunächst mit dem in Java integrierten Zlib-Dekompressor (java.util.zip.Inflater) extrahiert. Besonders bei der Client-Server-Kommunikation werden Nachrichten ab einer bestimmten Größe fast immer komprimiert gesendet. Als nächstes wird der Typ der Nachricht identifiziert. Handelt es sich beispielsweise um eine „Hello“-Nachricht, wird nun automatisch eine „Hello Answer“-Nachricht über den Stream zurückgeschickt. Wird eine Schlüsselnachricht aus Sicht einer Extraktionsroutine empfangen, wird entsprechend der Routine gehandelt. Meistens wird dann die Verbindung zum Client beendet, weil die nötigen Informationen erhalten wurden und das Aufrechterhalten der Verbindung nicht mehr notwendig ist.

Neben dem „Client Listener“-Thread für eingehende Client-Verbindungen wird zudem ein Thread für eingehende Nachrichten des gegenwärtig verbundenen Servers gestartet. Er funktioniert im Prinzip analog, d.h. er interpretiert die Nachrichten des Servers und reagiert entsprechend.

Über die Methoden der Klasse Connection kann eine Routine außerdem Nachrichten an andere Hosts versenden sowie die Serververbindung und den „Client Listener“ zusammen oder getrennt voneinander beenden.

Während konventionelle Clients es vorsehen, immer nur *eine* Verbindung zu einem Server aufrecht zu halten, ist mit dem entwickelten Programm auch eine beständige Verbindung zu vielen Servern gleichzeitig möglich, indem die Klasse Connection mehrfach instanziert wird, mit verschiedenen Serveradressen als Parameter.

Es ist uns nun möglich, eine passive aber dennoch dauerhafte Verbindung zum eDonkey/eMule-Netzwerk herzustellen. Der weitere Verlauf wird durch eine der Extraktionsroutinen bestimmt (Server bzw. Client Clusterer), die im Folgenden beschrieben werden.

5 Server Clustering

5.1 Vorgehensweise

5.1.1 Idee

Jeder eDonkey-Server sendet unmittelbar nach erfolgreichem Handshake neben der „ID Change“-Nachricht und den Begrüßungsnachrichten auch seinen Status. In dieser Nachricht stehen die momentane Anzahl von Clients, zu denen der Server eine Verbindung aufrecht hält, sowie die Anzahl verschiedener Dateien, die er (über seine Clients) kennt. Anhand dieser Informationen, welche Aussagen über die Größe des Servers machen, wollen wir die Ähnlichkeit zwischen Servern definieren. Ähnliche Server sollen dann identifiziert werden um anhand der resultierenden Cluster Muster in der Serverstruktur des eDonkey/eMule-Netzwerks zu identifizieren. Dabei sollen so viele Server wie möglich berücksichtigt werden.

Um nun an weitere Server-IP-Adressen zu gelangen, bietet das eDonkey-Protokoll einen Mechanismus an, mit dem ein Server auf Anfrage eine Liste mit allen ihm bekannten Serveradressen liefert [KB05]. Diese sollen nun auf die gleiche Weise bearbeitet werden um so eine möglichst vollständige Analyse sämtlicher eDonkey/eMule-Server gewährleisten zu können.

5.1.2 Implementierung

Die Server-Clusterer-Routine versucht (wie in 4.2 beschrieben) zuerst, eine Verbindung zum vorgegebenen Server herzustellen. Dann wartet sie auf eine „Server Status“-Nachricht, welche periodisch, zuerst allerdings immer direkt nach erfolgreichem Handshake, übermittelt wird. Der Status wird nun in einem Vektor zur späteren Berarbeitung durch den Clusteringalgorithmus abgespeichert. Es wurden Datenstrukturen dynamischer Größe gewählt, da vor einer derartigen Analyse nie bekannt ist, wie viele Server erreicht werden und da die Performance in der Größenordnung von maximal ein paar hundert Servern noch keine große Rolle spielt.

Man kann den Status eines Servers auch mit einer niedrigen ID abrufen und somit einen „Client Listener“-Thread für eingehende Client-Verbindungen auslassen, jedoch ist dann die Wahrscheinlichkeit höher, dass man vom Server aufgrund des erreichten Soft Limits abgestoßen wird (siehe 3.2) und dieser dann in der darauf folgenden Analyse fehlt. Daher ist eine hohe ID für unsere Zwecke unverzichtbar.

Um weitere Serveradressen zu erlangen, wird eine „Get Serverlist“-Nachricht an den Server gesendet. Die in der Antwortnachricht enthaltene Liste von IPs wird vom Programm berarbeitet, indem noch nicht bekannte IP-Adressen der lokalen Liste von IPs hinzugefügt und bereits bekannte ignoriert werden.

Eine Möglichkeit wäre nun, den Status jedes neuen Servers via UDP abzufragen [Kli04]. Jedoch bekämen wir dann *nur* den Status, und keine Liste mehr mit weiteren Server-IPs. Da wir aber möglichst viele Server in die Analyse mit einbeziehen wollen, verbinden wir uns nun via TCP mit jedem der lokalen IP-Liste neu hinzugefügten Server. Dies geschieht simultan über mehrere Instanzen desselben Threads. Immer wenn eine neue Liste von IP-Adressen empfangen wurde, wird sie auf bisher nicht bekannte Adressen untersucht und eine Reihe von Threads versuchen unverzüglich sich mit den neuen Servern zu verbinden um sie zu bearbeiten. Die Anzahl der simultanen Verbindungen wird dabei begrenzt durch einen vor Programmstart vom Benutzer vorgegebenen Portbereich.

Es ist hierbei darauf zu achten, dass die Threads für das Warten auf eingehende Nachrichten des verbundenen Endsystems stillgelegt werden, um nicht unnötig Ressourcen zu verbrauchen. Dazu wird das Konzept der Thread-Synchronisation verwendet [Ull04]. Sobald ein Thread eine Nachricht von seinem Server angefordert hat, wird er mit der „Wait“-Methode der Klasse Object in den passiven Zustand versetzt. Erst wenn die zugehörige Instanz des „Message Listeners“ die erwartete Nachricht über den Stream empfängt, benachrichtigt sie den wartenden Thread mit „Notify“, woraufhin dieser weitermachen kann. Wurde die Nachricht nach einer (einem Parameter entsprechenden) festgelegten Zeitspanne immer noch nicht empfangen, endet der Thread mit dem Fehlerfall des Timeouts.

Grundsätzlich wird zwischen den folgenden Fehlerfällen bei der Bearbeitung eines Servers unterschieden:

- Die TCP-Verbindung zum Server konnte nicht hergestellt werden.
- Ein Timeout liegt vor, da der Server in einer bestimmten Zeit nicht die erwartete Nachricht gesendet hat.
- Die Verbindung wurde unerwartet getrennt oder der Server ist voll (Soft oder Hard Limit) oder die IP des Clients wird blockiert.
- Es befinden sich Inkonsistenzen im Status des Servers.

Der zuerst beschriebene Fehlerfall deutet auf einen ungültigen oder inaktiven Server hin und schränkt daher die Vollständigkeit der Analyse nicht ein. Die anderen jedoch weisen sehr wohl auf funktionstüchtige Server hin, die aus bestimmten Umständen nicht erfolgreich erfasst werden konnten. Um das Ausmaß solcher Umstände zu minimieren, wurde ein Mechanismus implementiert, der eine Verbindungswiederholung im Falle eines Fehlschlags durchführt. Der Benutzer hat dabei die Möglichkeit anzugeben, wie oft ein Server in den oben beschriebenen Fehlerfällen wiederholt kontaktiert werden soll, wobei nach jedem gescheiterten Versuch zunächst eine variable Zeitspanne gewartet wird.

Immer wenn ein Server eine Liste von IP-Adressen schickt, sendet er auch eine „Server Identification"-Nachricht, welche den Namen des Servers und eine kurze Serverbeschreibung enthält. Diese Informationen werden ebenfalls gespeichert um später im Ergebnis den Servern Namen zuordnen zu können. Erst wenn alle drei benötigten Nachrichten (Server Status, Server Identifikation und die IP Liste der bekannten Server) vom Server empfangen wurden, wird der „Client Listener" beendet und somit der lokale TCP-Port für eingehende Client-Verbindungen wieder für die nächsten Threads freigegeben.

Bevor der Server nun in die Clusteranalyse mit einbezogen wird, wird er noch dem Prozess der Vorverarbeitung unterzogen. Hierbei werden Server mit Inkonsistenzen oder fehlenden Daten aussortiert, da diese zu Verzerrungen in den gefundenen Mustern führen würden. Wurden keine Inkonsistenzen gefunden, geht der Server in das Clustering mit ein.

Nachdem alle gefundenen Server bearbeitet wurden, werden die Zustände der erfolgreich erfassten, nachdem sie in das nötige Format gebracht wurden, dem Clusteringalgorithmus übergeben.

5.2 Clustering

5.2.1 Die Distanzfunktion

Das Ziel aller Clusteringalgorithmen ist es, möglichst ähnliche Objekte durch denselben Cluster zu repräsentieren. Objekte verschiedener Cluster hingegen sollen möglichst unähnlich sein [ES00]. Je nach Fragestellung bieten sich dabei verschiedene Modellierungen von Ähnlichkeit an. Ähnlichkeit wird jedoch immer durch eine Distanzfunktion für Paare von Objekten o_1, o_2 ausgedrückt und stützt sich dabei auf deren direkte oder abgeleitete Eigenschaften. Es gilt: Je größer die resultierende Distanz, desto unähnlicher die Objekte.

Für eine Distanzfunktion $dist$ müssen mindestens die folgenden Bedingungen gelten:

$dist(o_1, o_2) = d \in$
$dist(o_1, o_2) = 0$ genau dann wenn $o_1 = o_2$
$dist(o_1, o_2) = dist(o_2, o_1)$ (Symmetrie)

Die Ähnlichkeit zweier Server wird bei unserer Analyse durch die Distanz ihrer Größen $G_1 = (u_1, v_1)$ und $G_2 = (u_2, v_2)$ mit $u_1, v_1, u_2, v_2 \in {}^{0}$ zueinander bestimmt, wobei u_1, u_2 den Benutzeranzahlen und v_1, v_2 den Dateianzahlen der zwei Server entsprechen. Für die Ähnlichkeit zweier Datensätze mit numerischen Attributswerten eignet sich die *euklidische Distanz*, die sich hier wie folgt berechnet:

$$dist(G_1, G_2) = \sqrt{(u_1 - u_2)^2 + (v_1 - v_2)^2}$$

5.2.2 Der Clusteringalgorithmus

Bei der Clusteranalyse unterscheidet man grundsätzlich zwischen partitionierenden und hierarchischen Verfahren, wobei ein partitionierendes Verfahren eine Datenmenge in eine bestimmte Anzahl Cluster zerlegt, während ein hierarchisches Verfahren vielmehr eine hierarchische Repräsentation der Datenmenge erzeugt [ES00]. Bei den partitionierenden Verfahren enthält jeder Cluster mindestens ein Objekt und jedes Objekt ist genau einem Cluster zugeordnet.

Für das Clustering der Server wurde die klassische k-means-Methode verwendet, da sie die bekannteste und am häufigsten angewandte partitionierende Clusteringmethode darstellt. Die Voraussetzung für k-means ist, dass die Objekte Punkte $p = (x_1^p, ..., x_n^p)$ ($n \in$) in einem n-dimensionalen Raum sind. Für die Modellierung der Ähnlichkeit wird dabei die euklidische Distanz verwendet.

Bei k-means wird zu jedem der Cluster ein virtueller zentraler Punkt konstruiert. Ein solcher Punkt $\overline{x}_{C_z}$, auch Centroid genannt, ist der Mittelwert aller Punkte seines Clusters C_z:

$$\overline{x}_{C_z} = (\overline{x}_1(C_z), \overline{x}_2(C_z), ..., \overline{x}_n(C_z))$$

wobei $\overline{x}_j(C_z) = \frac{1}{|C_z|} \cdot \sum_{p \in C_z} x_j^p$ mit $1 \leq j \leq n$ der Mittelwert für die j-te Dimension der Punkte im Cluster C_z ist.

Ziel des Clusteringalgorithmus soll es nun sein, die Varianz innerhalb eines Clusters zu minimieren. Dazu muss die Summe der quadrierten Distanzen jedes Punktes zum Centroid seines Clusters so klein wie möglich sein. Diese Summe wird auch Kompaktheit $TD^2 = \sum_{i=1}^{k} TD^2(C_i)$ eines Clusterings genannt, wobei $TD^2(C_i) = \sum_{p \in C_i} dist(p, \overline{x}_{C_i})^2$ die Kompaktheit eines einzelnen Clusters C_i und k die Anzahl der bestehenden Cluster ausdrückt.

In dem entwickelten Programm lässt sich die Kompaktheit des Clusterings mit einer Methode berechnen und ausgeben. Der k-means-Algorithmus wurde wie folgt implementiert:

```
k-means-Algorithmus(Punktmenge, k)
   Erzeuge zufällig eine Zerlegung der Punktmenge in
   k Klassen;
   Berechne die Menge C' der Centroide für die
   k Klassen;
   C'' = {};
   repeat until C'' = C'
     C'' = C';
     Bilde k Klassen durch Zuordnung jedes Punktes
     zum nächstliegenden Centroid und passe jeweils
```

```
    die Centroide des alten und neuen Clusters
    inkrementell an;
    Bilde C', die Menge der resultierenden Centroide;
  return C'';
```

Bei der obigen Methode werden die Punkte also so lange den Clustern ihrer nächsten Centroide zugeordnet, bis schließlich keine Änderung mehr eintritt. Immer wenn ein Punkt p von Cluster C_1 zu Cluster C_2 wechselt, werden die Koordinaten der zugehörigen Centroide folgendermaßen inkrementell angepasst:

$$\overline{x}_j(C_1') = \frac{1}{|C_1| - 1}(|C_1| \cdot \overline{x}_j(C_1) - x_j^p)$$

$$\overline{x}_j(C_2') = \frac{1}{|C_2| + 1}(|C_2| \cdot \overline{x}_j(C_2) + x_j^p)$$

5.2.3 Laufzeit und Speicherbedarf

Der in 5.2.2 beschriebene Algorithmus weist in der Theorie sowie in der Implementierung in Abhängigkeit der Objektanzahl n einen Aufwand von $O(n)$ für eine Iteration auf, denn dabei wird immer für jedes Objekt (also n Mal) die Distanz zu den aktuellen Centroiden berechnet und das Objekt evtl. neu zugeordnet. Die Anzahl der Iterationen beträgt dabei im Allgemeinen nur ca. fünf bis zehn.

Die Größe des benötigten Speichers ist zu vernachlässigen, denn es wird außer der Datenmenge selbst noch ein Array der Größe n vom Typ Integer für die Clusterzuordnung der Objekte benötigt. Wenn die k-means-Methode nicht benutzt wird, d.h. die Centroide erst aktualisiert werden, sobald alle Objekte zugeordnet wurden, fällt ein weiteres Array der gleichen Größe für die zukünftige Clusterzuordnung an. Zudem werden die Centroide in einem Double-Array der Größe *Clusteranzahl * Dimension* gespeichert. Sollen z.B. 1000 Server unter der Berücksichtigung von zehn Dimensionen (des Typs Integer) in fünf Cluster unterteilt werden, beträgt der zusätzliche Speicherbedarf nur ca. 8,2 kb.

5.2.4 Der Silhouetten-Koeffizient

Ein Problem bei partitionierenden Clusteringverfahren ist, dass die Anzahl der zu bestimmenden Cluster vorgegeben ist und infolgedessen unter Umständen eine schlechte Struktur erzwungen wird. Dadurch können Cluster mit kaum ähnlichen Objekten entstehen bzw. dichte Cluster gar nicht identifiziert werden. Auch Rauschen in der Datenmenge kann bei einer vorgegebenen Clusteranzahl zu unerwünschten Effekten in der resultierenden Clusterstruktur führen. Daher ist ein Maß für die Güte eines Clusterings vorteilhaft, welches unabhängig von der Anzahl Cluster ist. Wenn wir

dann für jede mögliche Anzahl Cluster ein Clustering erzeugen lassen und jedes Mal das Maß bestimmen, können wir daran die beste Anzahl Cluster ermitteln. Ein solches Maß stellt der *Silhouetten-Koeffizient* dar.

Sei o ein Objekt aus einer Objektmenge O und C_f der zu o gehörige Cluster, der wiederum Teil der Gesamtmenge von Clustern $C = \{C_1, ..., C_k\}$ mit $1 \leq f \leq k$ ist, also $o \in C_f \in C$. Nach [ES00] ist der durchschnittliche Abstand von o zu „seinem" Cluster C_f:

$$a(o) = \frac{\sum_{p \in C_f} dist(o,p)}{|C_f|}$$

Der durchschnittliche Abstand des Objekts o zu seinem „Nachbarcluster" C_g (dem Cluster, zu dem es außer seinem eigenen die niedrigste Distanz hat) ist demnach:

$$b(o) = min_{C_g \in C, C_g \neq C_f} \frac{\sum_{p \in C_g} dist(o,p)}{|C_g|}$$

Die *Silhouette* eines Objekts $o \in C_f$ gibt an, wie gut dieses klassifiziert ist:

$$s(o) = \begin{cases} 0 & \text{wenn } |C_f| = 1 \\ \frac{b(o)-a(o)}{max\{a(o),b(o)\}} & \text{sonst} \end{cases}$$

Im besten Fall nimmt die Silhouette den Wert 1 an. Dann ist die Distanz zum eigenen Cluster minimal (also 0) und die zum Nachbarcluster größer. Im schlechtesten Fall jedoch hat sie den Wert -1, was eine minimale durchschnittliche Distanz zum Nachbarcluster bedeutet.

Die *Silhouettenweite* eines Clusters C_i ergibt sich nun aus der durchschnittlichen Silhouette all seiner Objekte:

$$s(C_i) = \frac{\sum_{p \in C_i} s(p)}{|C_i|}$$

Der Silhouetten-Koeffizient eines Clusterings C berechnet sich aus der durchschnittlichen Silhouettenweite aller Cluster, oder auch der durchschnittlichen Silhouette *aller* Objekte:

$$s(C) = \frac{\sum_{C_i \in C} \sum_{p \in C_i} s(p)}{|O|}$$

Er kann demnach maximal den Wert 1 annehmen, wobei die Struktur gegen 1 hin immer stärker wird und bei Werten unter 0,25 nicht mehr von einer Struktur zu sprechen sein kann.

Das Programm führt nun also den k-means-Algorithmus für $k = 2, ..., n-1$ durch und teilt der aufrufenden Routine das k mit, für das der Silhouetten-Koeffizient maximal ist. Es wird ebenfalls über das zweitbeste k informiert. Für ein gewünschtes k kann der Routine dann ein Array mit Clusterzuweisungen für jedes Objekt zur Ausgabe der Ergebniscluster übergeben werden.

5.3 Ergebnisse

Die Server-Clusterer-Routine des Programms wurde an verschiedenen Tagen zu unterschiedlichen Tageszeiten ausgeführt. Die gewählten Parameter sahen dabei maximal neun Verbindungswiederholungen (nach vorangehenden Wartezeiten von je drei Minuten) vor, damit auch volle Server evtl. doch noch erreicht würden. Ein zusätzliches Erhöhen der Parameter hat keine Verbesserung bezüglich der Anzahl der erreichten Server zeigen können.

In jedem der Durchläufe konnten zwischen 98 und 106 Server erfolgreich kontaktiert werden. Im Folgenden wird ein repräsentatives Ergebnis mit vergleichsweise wenig Fehlerfällen aufgeführt:

```
Statistik:
106 Server von 157 erfolgreich bearbeitet.
44: konnte Verbindung nicht herstellen.
2: Timeout bei Warten auf Servernachricht.
0: inkonsistenter Serverstatus.
5: Server ist voll bzw. Verbindung verloren.
```

Bei all jenen Servern, zu denen keine Verbindung hergestellt werden konnte, ist nicht zwangsläufig von gültigen eDonkey-Servern auszugehen, da solche *jeden* Verbindungswunsch annehmen, selbst wenn der Client anschließend abgewiesen wird, weil z.B. der Server voll ist. Diese Aussage wird dadurch bekräftigt, dass in der Liste der nicht erreichten Server zwölf Mal eine Adresse aus dem Bereich 127.0.0.0/8 zu finden ist. Diese ist keine gültige Internet-IP, sondern stammt aus einem Adressbereich, der für besondere Verwendung reserviert ist (der Loopback-Adresse 127.0.0.1) [TNWG02].

Wenn wir die Benutzeranzahlen sämtlicher Server aufaddieren, kommen wir auf 6345364 Clients, die mit dem eDonkey-Netzwerk verbunden waren, zuzüglich solcher, deren Server nicht erreicht werden konnte. Es ist aber davon auszugehen, dass zudem private Server existieren, deren IP-Adressen den öffentlichen Servern nicht bekannt sind. Die Anzahl der zu diesen Servern verbundenen Benutzer wird jedoch genau aus diesem Grund verhältnismäßig klein sein. Insofern sollte der Großteil erfasst sein und die Zahl somit eine realisitische Schätzung darstellen.

Beim Berechnen der Clusterings für $k = 2, ..., 105$ stellt sich jenes mit zwei Clustern (siehe Abbildung 1) als das qualitativ beste heraus. Der Silhouetten-Koeffizient beträgt ca. 0,937. Das Clustering weist also eine sehr starke Struktur auf. Diese Tatsache, zusammen mit einer relativ geringen Anzahl von nicht erreichbaren gültigen Servern, lassen darauf schließen, dass das Ziel einer Musterfindung in der Serverstruktur des eDonkey-Netzwerks erfolgreich erreicht werden konnte.

Der erste der beiden Cluster enthält zwei Server und hat einen Centroid von ca. (733292, 96159292), wobei der erste Wert der durchschnittlichen Be-

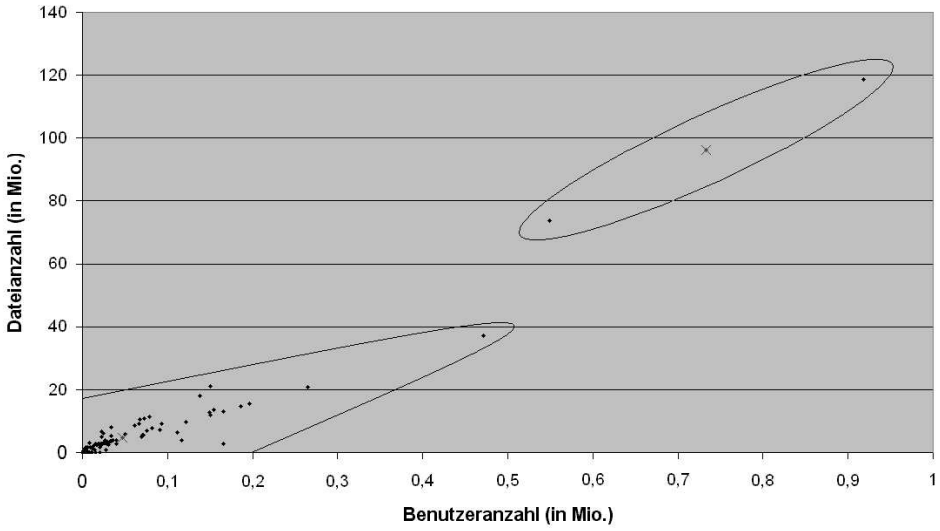

Abbildung 1: Server Clustering mit k = 2

nutzeranzahl und der zweite der mittleren Dateianzahl sämtlicher Server des Clusters entspricht. Der Centroid des zweiten Clusters hingegen beträgt ca. (46911, 4654597) und ergibt sich aus den Daten von 104 Objekten. Es lässt sich also sagen, dass zum Zeitpunkt der Erhebung zwei dominante Server existierten, die im Durchschnitt 15 Mal größer als die restlichen Server waren und sich somit deutlich von diesen abgrenzten. Der größte Server, der laut RIPE-WHOIS-Datenbank (http://www.ripe.net/) in Belgien steht, hatte zum Zeitpunkt der Erhebung 917977 Client-Verbindungen. Das sind ca. ein Siebtel *aller* aufgedeckten Clients. Der andere Server des Clusters ist niederländisch und bediente ca. ein Elftel der gefundenen Clients. Das bedeutet, dass allein zu diesen beiden Servern beinahe 25% aller erfassten Peers verbunden waren.

Die Existenz zweier herausragender Server ist zwar eine interessante Entdeckung, hinterlässt den Großteil der Server aber immer noch als eine Einheit ohne jegliche Muster. Es bietet sich daher an, das Clustering mit dem zweitbesten Silhouetten-Koeffizienten zu betrachten (siehe Abbildung 2). Dieser beträgt hier allerdings „nur noch“ ca. 0,731.

Das Clustering enthält drei Cluster und unterscheidet sich dadurch vom zuerst betrachteten, dass es in der Verteilung der 104 Server einen weiteren Cluster identifiziert. Der objektreiche Cluster wird dabei nämlich unterteilt in einen mit 18 „mittelgroßen“ Servern (Centroid ca. (153024, 14493811)) und einen mit 86 „kleinen“ Server (Centroid ca. (24700, 2595227)). Wir können daraus schließen, dass der Großteil (ca. 80%) aller eDonkey-Server im Durchschnitt jeweils 24700 Clients bedient und 2595227 Dateien über seine Clients anbietet.

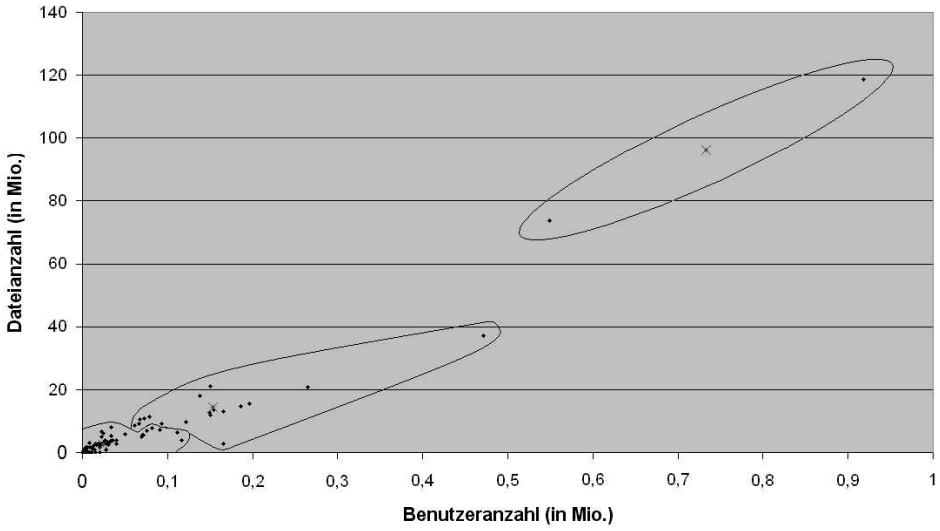

Abbildung 2: Server Clustering mit k = 3

Eine Statistik über die Standorte der erfolgreich erfassten Server stellt Abbildung 3 dar. Die Informationen wurden dabei mit Hilfe des IP-Adress-Services der IANA gewonnen (http://www.iana.org/ipaddress/ip-addresses.htm).

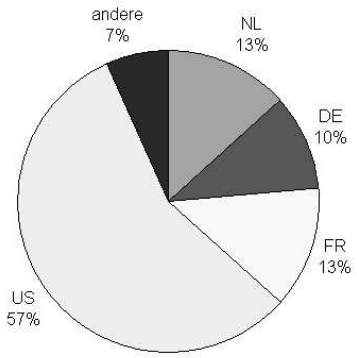

Abbildung 3: Server-Standorte

5.4 Ausblick

Bisher wurden Server nur unter Berücksichtigung von zwei Dimensionen geclustert, da bei den betrachteten Eigenschaften (durch das Protokoll) sichergestellt war, dass diese Informationen von jedem erfolgreich kontaktierten Server bezogen werden können. Es sind jedoch durchaus weitere

Eigenschaften denkbar, die den Ähnlichkeitsbegriff von Servern erweitern. Das eDonkey-Protokoll sieht z.B. in der „Status Response“-Nachricht des UDP-Nachrichtensatzes Felder für die maximale Anzahl von Dateien vor, die ein Server in seine Datenbank aufnehmen kann. Diese Felder sind jedoch optional und würden somit in der Analyse diverse Server ausschließen. Weiterhin ließen sich die Herkunftsländer der Server als zusätzliche Dimension darstellen, deren Gleichheit bzw. Ungleichheit in die Berechnung der Distanzen mit einbezogen werden könnte. Die Länder müssten dann allerdings anhand der IPs im Programmablauf dynamisch ermittelt werden. Auch über die Antwortzeiten der Server, die man als Indiz für ihre derzeitige Belastung auffassen könnte, ließe sich Ähnlichkeit definieren. Die Distanzfunktion im entwickelten Programm wurde daher so implementiert, dass sie beliebig austauschbar ist und Raum für weitere Ähnlichkeitsdefinitionen bietet.

Das in 5.3 präsentierte Clustering ist das Ergebnis eines ausgesuchten Clusteringverfahrens und muss keine optimale Zerlegung der Menge von Servern darstellen. Hier bleibt daher Raum, die Fragestellung mit anderen Clusteralgorithmen zu untersuchen und die Ergebnisse miteinander zu vergleichen. Es bietet sich zum Beispiel das verwandte Medoid-Clustering an, in dem existierende Objekte (Medoide) als Mittelpunkte der Cluster betrachtet werden. Auch ein hierarchisches Clustering ist durchaus denkbar.

6 Client Clustering

6.1 Vorgehensweise

6.1.1 Idee

Das eDonkey-Protokoll stellt eine Funktion bereit, mit der ein Peer die Liste von Dateien eines anderen Peers einsehen kann [KB05]. Die Idee ist nun, diese Funktion zu verwenden, um Peers mit möglichst identischen Inhalten zu identifizieren.

Problematisch zeigt sich die Tatsache, dass die angesprochene Funktion von den meisten gängigen eMule-Clients (insbesondere dem offiziellen) standardmäßig deaktiviert ist und für die meisten Benutzer kein Anreiz dazu besteht, sie zu aktivieren und somit freiwillig ihr anonymes Netzwerkverhalten der Öffentlichkeit zu offenbaren. Es kann also nur ein verhältnismäßig kleiner Teil aller Peers berücksichtigt werden. Dennoch sollen so viele Peers wie möglich analysiert werden, wobei sich die Frage stellt, wie wir an möglichst viele IP-Adressen anderer Peers gelangen. Ein Server stellt nämlich keine Liste von Client-IPs bereit. Er antwortet lediglich auf Anfrage nach einer bestimmten Datei mit einer Liste von Quellen, die diese Datei anbieten. Es ist daher die Aufgabe, Dateien auszumachen, die möglichst viele Peers besitzen. Gleichzeitig gilt es sicherzustellen, dass man

nicht zu viele Anfragen in kurzer Zeit an den Server schickt. Dies würde nämlich in einer temporären Sperre der eigenen IP resultieren.

Zuerst wird daher an den Server eine möglichst effektive Suchanfrage gestellt: *Zeige alle Dateien, in deren Name ein „a" oder ein „b" oder ein „c" usw. vorkommt, und deren Mindestverfügbarkeit 200 Clients beträgt*. Der Server antwortet mit einer Liste von Dateien. Zu jeder Datei informiert er auch über die zugehörige Client-ID eines beliebigen Peers, der die Datei anbietet. Nun wird zu jeder gefundenen Datei in festgelegten Intervallen eine Anfrage an den Server nach weiteren Quellen geschickt. Dieser liefert jedes Mal eine ganze Liste von IPs der Clients zurück, die diese Datei anbieten. Zu jedem noch nicht bekannten Client dieser Liste wird dann unverzüglich Verbindung aufgenommen und nach der Liste seiner Dateien gefragt.

6.1.2 Implementierung

Nachdem die Client-Clusterer-Routine die Verbindung zum Server erfolgreich hergestellt hat (siehe 4.2), wird an diesen eine „Search Request"-Nachricht gesendet.

Das eDonkey-Protokoll bietet zwei Möglichkeiten für die Dateisuche an, die auf unterschiedlichen Transportprotokollen basieren [KB05]. Die Anfrage per UDP hat den Vorteil, dass keine Verbindung zum Server benötigt wird. Jedoch hat sich gezeigt, dass ein Server auf diese Weise, selbst bei weit verbreiteten Suchstrings, immer nur höchstens drei Ergebnisse liefert. Das kommt wahrscheinlich daher, dass er „fremden" Clients niedrigere Prioritäten einräumt als seinen eigenen (also denen, zu den er eine TCP-Verbindung hat). Es wird daher die Dateisuche über TCP verwendet.

Bei der Konstruktion der Suchanfrage muss ein binärer Baum erzeugt werden, in dem die verschiedenen Bedingungen (z.B. enthaltener String, Mindestverfügbarkeit, Mindestgröße) mit AND verknüpft sind. Die Zusammensetzung des Suchstrings stellt dabei einen weiteren binären Teilbaum mit den zusätzlichen Boolschen Operatoren OR und NOT dar. Die im Programm verwendetete Anfrage hat die in Abbildung 4 dargestellte Form und wird in Prefix-Reihenfolge übermittelt. Konventionelle eMule-Clients bieten in der Regel keine Unterstützung für derartig verschachtelte Anfragen.

Nachdem der Server mit einer „Search Result"-Nachricht geantwortet hat, wird er für jeden Treffer (maximal 300 pro Anfrage) in vorgegebenen Zeitintervallen mit einer „Get Sources"-Nachricht aufgefordert, die zugehörigen Client-Adressen zu übermitteln. Auch hier bieten sich wieder zwei Möglichkeiten:

Die Anfrage über UDP ist insofern wieder vorteilhaft, als dass direkt mehrere Dateien angegeben werden können. Jedoch liefert der Server hierbei meistens nicht mehr als zehn Quellen pro Datei zurück. Außerdem wird die lokale IP-Adresse blockiert wenn zu viele Dateien auf einmal angege-

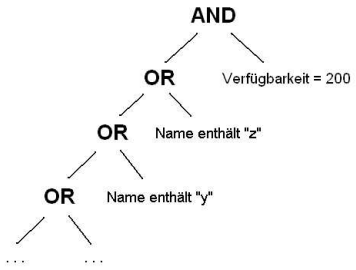

Abbildung 4: Die implementierte Suchanfrage

ben werden. Daher stellt sich diese Variante wieder als ungeeignet heraus. Quellenanforderungen per TCP dagegen liefern zwar nur Quellen für jeweils eine Datei, dafür ist die zurückgegebene Anzahl meistens um einiges höher.

Um die Clients zu erreichen, werden viele simultane Threads gestartet, die je einen Client bearbeiten. Die Threads brechen ab, sollte die vorgegebene zu clusternde Anzahl von Clients erreicht sein (sofern eine solche angegeben wurde). Außerdem warten die Threads, wenn gerade eine Liste von Dateien eines Clients bearbeitet wird, damit Clients mit vielen Dateien nicht eine geringere Chance haben, in die Analyse mit einbezogen zu werden. Wenn ein Thread Verbindung zu einem Client aufgenommen hat, wird der „Client Message Processor“ aktiv und der in 3.3 beschriebene Handshake findet statt. Sobald die „Hello Answer“-Nachricht empfangen wurde, wird dem Client eine „View List Of Shared Files“-Nachricht zugesandt. Nun sollte der Client mit einer „View List Of Shared Files Answer“-Nachricht antworten. Danach kann die Verbindung zum Client beendet und der Inhalt der Nachricht extrahiert werden. Ist die Trefferanzahl der Nachricht 0, dann wird die Funktion der Dateieinsicht vom Client nicht unterstützt. Ist das Ergebnis größer als 0, folgt die entsprechende Liste von Dateien, die pro Datei die ID, den Dateinamen, die Dateigröße sowie diverse optionale Tags enthält, wie z.B. Dateiformat, Codec, Album, Interpret usw.

Die Schwierigkeit bei der Implementierung bestand darin, den Datentyp des jeweiligen Tags zu identifizieren. Beispielsweise folgen einem Tag des Typs 0x82 zwei Bytes, in denen die Größe des String kodiert sind, ein Tag des Typs 0x13 steht dagegen immer für einen drei Byte großen String. Das allgemeine Format von Tags des eDonkey-Protokolls kann in [HB02] nachgelesen werden.

Dateien werden im eDonkey/eMule-Netzwerk nicht nach ihrem Namen identifiziert. Stattdessen wird anhand ihres Inhalts ein 128-Bit-„Fingerabdruck“ berechnet, die sogenannte Datei ID. Dazu wird der MD4-Algorithmus verwendet (siehe [TNWG92]). Die Datei IDs und die zuge-

hörigen Dateinamen werden also herausgefiltert und abgespeichert. Dabei werden, anders als beim Clustering der Server, keine Vektoren benutzt, da Datenstrukturen dynamischer Größe vergleichsweise langsam sind, besonders bei Clients, die z.B. 20000 Dateien anbieten und infolgedessen 20000 * 16 Bytes dem Vektor hinzugefügt werden sollen. Es wird für die Datei IDs daher ein statisches Array verwendet, welches zu Beginn mit einer maximalen Dateianzahl vorinitialisiert werden muss. Die Dateinamen werden aus Effizienzgründen gar nicht im Hauptspeicher gehalten, sondern in eine externe Datei gespeichert. Die Namen der Dateien, die alle Clients eines Clusters (im Durchschnitt) besitzen, werden dann am Ende wieder aus der Datei ausgelesen.

Konnte ein Client nicht erreicht werden oder liegt ein Timeout vor, wird auch hier wieder eine (dem Parameter entsprechende) Anzahl Wiederholungsversuche gestartet. Es wird dabei zwischen den folgenden Fehlerfällen bezüglich der Bearbeitung eines Clients unterschieden:

- Der Client verweigert die Einsicht in seine Dateiliste.
- Der Client hat eine niedrige ID und kann nicht erreicht werden.
- Die Verbindung zum Client konnte nicht hergestellt werden.
- Ein Timeout liegt vor, da der Client in einer bestimmten Zeit nicht die erwartete Nachricht gesendet hat.
- Die Verbindung wurde unerwartet getrennt (wahrscheinlich aufgrund zu vieler Dateien).

Wenn alle Dateien der „Search Result“-Nachricht abgearbeitet wurden, wird eine „More Results“-Nachricht an den Server geschickt. Er wird hiermit instruiert, weitere Treffer bezüglich der vorangegangenen Suchanfrage zurückzugeben, vorausgesetzt es gibt solche.

Nun wird für die neuen Dateien wieder nach Quellen gefragt usw. Erst wenn der Server keine weiteren Ergebnisse mehr liefert oder wenn die gewünschte Anzahl Clients erreicht ist, werden dem Clusteralgorithmus die gesammelten Datei IDs übergeben.

6.2 Clustering

6.2.1 Die Distanzfunktion

Wie beim Clustering der Server muss auch hier bei der Wahl der Distanzfunktion berücksichtigt werden, wie man mit den gegebenen Variablen, also Listen von Dateien (eigentlich Datei IDs), Ähnlichkeit ausdrücken möchte.

Es seien zwei Peers gegeben, die jeweils eine endliche Menge von Dateien $\mathcal{L}_1 = (a_1, ..., a_m)$ und $\mathcal{L}_2 = (b_1, ..., b_n)$ mit $a_1, ..., a_m, b_1, ..., b_n \in$ anbieten. Wir definieren die Distanz zwischen diesen Peers über die Distanz zwischen ihren Dateilisten. Eine geeignete Distanzfunktion, welche (nach [ES00]) den Anteil der verschiedenen Elemente in $\mathcal{L}_1$ und $\mathcal{L}_2$ berechnet, lautet:

$$dist(\mathcal{L}_1, \mathcal{L}_2) = \frac{|\mathcal{L}_1 \cup \mathcal{L}_2| - |\mathcal{L}_1 \cap \mathcal{L}_2|}{|\mathcal{L}_1 \cup \mathcal{L}_2|}$$

Die Ähnlichkeit zweier Clients hängt also sowohl von der Anzahl der Dateien im Durchschnitt, als auch von der Anzahl der vereinigten Dateien ab. Dies bedeutet, dass zwei Clients mit nur wenigen identischen Dateien sich ähnlicher sein können als solche mit einer größeren Anzahl gleicher Dateien - nämlich dann, wenn sie in der Vereinigung viel weniger Dateien anbieten.

6.2.2 Der Clusteringalgorithmus

Bei der Wahl des Clusteringverfahrens kommt das beim Clustering der Server verwendete Centroid-Clustering nicht in Frage, da die (in 5.2.2 beschriebenen) Voraussetzungen nicht erfüllt sind. Die zu betrachtenden Eigenschaften sind nicht numerischer Natur und stellen keine Punkte in einem Raum mehr dar. Vielmehr handelt es sich um Listen von Dateien, d.h. Variablen nominalen Skalenniveaus, deren Ausprägungen keine Rangordnung besitzen. Daher kann man hier auch keinen Mittelwert mehr bestimmen, sondern lediglich Transformationen durchführen, die auf Verschiedenheit bzw. Gleichheit der Daten basieren [BBPW03]. Es wird daher ein hierarchischer Clusteringalgorithmus verwendet.

Wie bereits erwähnt, erzeugen hierarchische Clusteringverfahren eine hierarchische Repräsentation der Daten. Diese werden durch ein sogenanntes „Dendrogramm“ repräsentiert [ES00]. Ein Dendrogramm ist ein Baum, der die Zerlegung der Datenmenge in immer kleinere Teilmengen (Cluster) darstellt. Dabei ist die Wurzel der Cluster, der alle Objekte enthält und die Blätter entsprechen einelementigen Clustern bzw. den einzelnen Objekten. Jeder Knoten des Baums repräsentiert einen Cluster, der sämtliche Objekte aller Cluster im Teilbaum des Knotens enthält.

Üblicherweise wird agglomerativ vorgegangen, das heisst das Dendrogramm wird "bottom-up"(von unten nach oben) aufgebaut. Der Algorithmus dazu lautet:

1. Bestimme die Distanzen zwischen allen Paaren von Objekten.

2. Bilde einen neuen Cluster aus den zwei Objekten oder Clustern, welche die geringste Distanz zueinander haben.

3. Bestimme die Distanz zwischen dem neuen Cluster und allen anderen Objekten und Clustern (die übrigen Distanzen bleiben unverändert).

4. Wiederhole ab Schritt 2, bis sich alle Objekte in einem einzigen Cluster befinden.

Bei dem obigen Algorithmus müssen ab Schritt 2 auch Distanzen zwischen zwei Clustern bestimmt werden. Hierbei stellt sich die Frage, wie man den Abstand zu einem Cluster definiert: Bei der „Single-Link"-Variante ist dieser gleich dem Abstand zum nächsten Objekt des Clusters. Die „Complete-Link"-Methode sieht das Gegenteil vor, also den Abstand zum weitesten entfernten Objekt. Bei „Average-Link" entspricht die Distanz zu einem Cluster der durchschnittlichen Distanz zu allen Objekten dieses Clusters. Im entwickelten Programm wurde die gängige Single-Link-Variante implementiert.

6.2.3 Laufzeit und Speicherbedarf

Da hier im Gegensatz zum Server Clustering unter Umständen mit tausenden von Objekten gearbeitet wird, ist besonders auf eine effiziente Implementierung zu achten. Die Komplexität des in 6.2.2 beschriebenen Clusteringverfahrens in Abhängigkeit der Anzahl einbezogener Objekte n ist $O(n^2)$, denn es müssen (in Form einer Matrix) Distanzen für jedes Paar von Objekten berechnet werden. Danach wird $n - 2$ Mal die beste Distanz in der Matrix gesucht (ebenfalls $O(n^2)$) und die betroffenen zwei Cluster mit anschließender Neubestimmung der Distanzen des neuen Clusters zu den restlichen agglomeriert.

Aus Effizienzgründen werden bei der Implementierung die initialen Distanzen zwischen allen Objekten im Speicher behalten, damit bei jeder zukünftigen Distanzbestimmung auf sie zurückgegriffen werden kann. Andernfalls müssten sämtliche Distanzen nach jeder Agglomeration neu berechnet werden, was einen Gesamtaufwand von $O(n^3)$ hieße. Dies hat jedoch auch einen nachteiligen Effekt was den benötigten Speicherplatz betrifft, denn ein Array, das für beispielsweise 10000 Clients sämtliche Distanzen (des Datentyps Double) enthält, benötigt ca. 381,4 MB Speicher.

Von der Matrix, die für die Distanzen zwischen den Clustern verwendet wird, ist nur knapp die Hälfte eigentlich belegt. Aus Speicherplatzgründen wird die Matrix daher als ein Array der Größe $\frac{n^2}{2} - \frac{n}{2}$ initialisert, was genau dem Dreieck unter der Diagonalen (die überall 0 enthält) entspricht und somit keinen redundanten Speicherplatz belegt.

Es wird zudem eine weitere Distanzmatrix mit der gleichen Größe für die sich stets ändernden Distanzen verwendet und ein Integer-Array der Größe n^2, um die aktuelle Clusterbelegung der Objekte zu speichern.

6.3 Ergebnisse

Aufgrund der geringen Wahrscheinlichkeit Peers zu finden, die auch nur wenige identische Dateien besitzen, erscheinen in der Praxis erst Analysen mit einer Vielzahl von Clients sinnvoll. Dennoch soll im Folgenden zur Veranschaulichung ein Clustering mit nur 15 Clients aufgegriffen werden. Die Parameter wurden dabei so gewählt, dass die Anzahl der Fehlerfälle möglichst minimiert wird. Es wurde keine Clusteranzahl vorgegeben, so dass das qualitativ beste bestimmt werden sollte. Die Statistiken über die nicht erreichten Clients fielen dabei wie folgt aus:

```
15 von 323 Clients erfolgreich bearbeitet.
138: Verweigerung der Dateieinsicht.
166: Niedrige ID.
4: konnte Verbindung nicht herstellen.
0: Timeout bei Warten auf Clientnachricht.
0: Verbindung verloren.
```

Wie sich an der Aufschlüsselung der Fehlerfälle ablesen lässt, befanden sich über die Hälfte der Clients hinter einer Firewall (bzw. blockierten ihren eDonkey-/eMule-Port). Von den übrigen ließen fast gar keine den Zugriff auf die Liste ihrer Dateien zu, so dass nur gut 5% aller Clients erfolgreich bearbeitet werden konnten. Bei verschiedenen Durchläufen mit unterschiedlichen Servern lag dieser Anteil ungefähr zwischen 5% und 10% und nur ca. 10% wiesen eine niedrige ID auf. Der Großteil (ca 70%) verweigerte die Dateieinsicht.

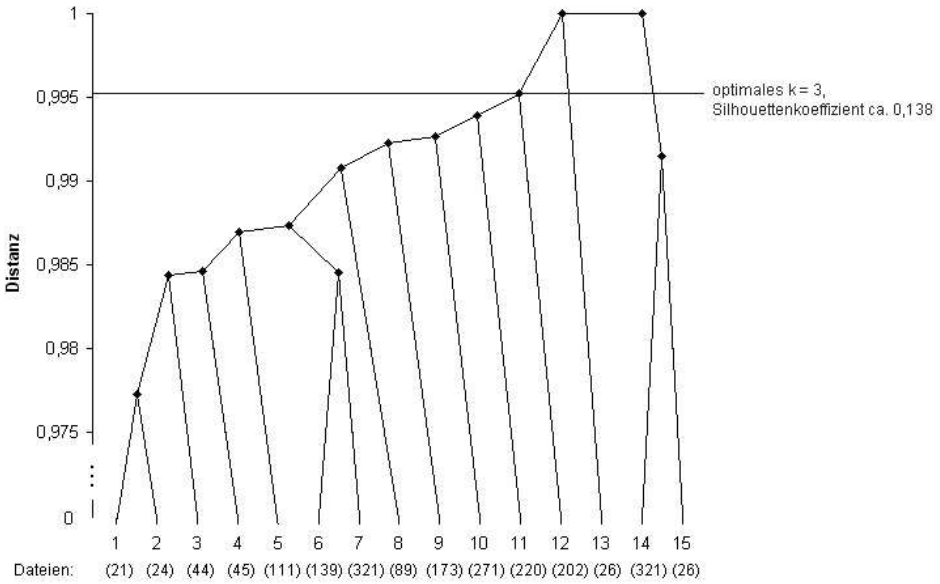

Abbildung 5: Dendrogramm

Im anschließenden Prozess des Clusterings informiert die Ausgabe des Programms über die erzeugte hierarchische Repräsentation der Cluster (siehe Abbildung 5). Dabei wird nach jeder Agglomeration der Silhouetten-Koeffizient neu berechnet, um die beste Anzahl Cluster bestimmen zu können. Wie man dem Dendrogramm entnehmen kann, findet die initiale Agglomeration erst bei einer Distanz von ca. 0,977 statt. Das bedeutet, die Dateien der beiden ähnlichsten Peers sind zu 97,7% verschieden. Insgesamt betrachtet lässt sich daher sagen, dass sämtliche der Analyse unterzogenen Peers kaum Ähnlichkeiten miteinander aufweisen. Dies führt zudem zu einem niedrigen Silhouetten-Koeffizienten. Bei $k = 3$ ist dieser maximal und beträgt lediglich ca. 0,138.

Von den drei Clustern der optimalen Zerlegung besteht einer aus zwei Peers, die zwei identische Dateien anbieten (sich aber trotzdem noch zu ca. 99,15% unterscheiden). Der andere Cluster besteht aus zwölf Objekten, die im Durchschnitt nur eine gemeinsame Datei anbieten. Die Tatsache, dass so viele Peers diese Datei besitzen, ist durch die Vorgehensweise des Programms begründet, denn es wurden ja Quellen zu der gleichen Datei abgefragt. Der dritte Cluster ist einelementig und hat zu beiden anderen Clustern die maximale Distanz 1. Der Peer hat also keine Datei mit den anderen 14 gemeinsam.

Es ist davon auszugehen, dass *immer* nur ein geringer Teil der einbezogenen Clients eine stärkere Ähnlichkeit aufweist, ungeachtet der Anzahl an involvierten Clients. Daher wird der Silhouetten-Koeffizient auch nie auf eine brauchbare Struktur hinweisen. Das heißt jedoch nicht, dass nicht optimal geclustert wurde und Clients mit Ähnlichkeiten nicht durch denselben Cluster repräsentiert werden. Der Silhouetten-Koeffizient sagt vielmehr etwas darüber aus, wie gut die Menge der Objekte sich in disjunkte Teilmengen mit möglichst ähnlichen Objekten unterteilen lässt.

Analysen in größerem Rahmen bestätigen, dass der Silhouetten-Koeffizient auch bei zunehmender Clientanzahl nicht größer wird. Tabelle 1 gibt einen Überblick über die erzeugten Clusterings für Analysen verschieden großer Ausmaße.

Clientanzahl	optimales k	Silh.koeffizient	größter Cluster
15	3	$\sim 0,138$	12
1073	785	$\sim 0,068$	98
3000	2501	$\sim 0,034$	200

Tabelle 1: Clusterings für 15, 1073 und 3000 Clients

6.4 Ausblick

Die implementierte Methode erlaubt es uns, eine Vielzahl von Clients des eDonkey-Netzwerks nach ähnlichen Dateiverzeichnissen zu clustern. In der Praxis konnten dabei bis zu 143823 Client-IPs in einem Durchlauf aufgedeckt werden, allerdings nur von Clients desselben Servers. Da wir für die Analyse aber möglichst viele Clients erreichen wollen, ließe sich das Programm dahingehend erweitern, dass Clients von mehreren Servern berücksichtigt werden. Es würde sich dabei um eine Verschmelzung der beiden implementierten Routinen handeln, bei der jeder auffindbare Server nach möglichst vielen Client-Adressen abgefragt wird.

Um weiterhin zusätzliche Clients erreichen zu können, ließe sich eine Unterstützung für Clients mit niedriger ID implementieren, so dass auch diese nach ihren Dateiverzeichnissen befragt werden können. Dazu müsste für jeden dieser Clients eine entsprechende Aufforderung an den Server gesendet werden, woraufhin dieser den Client zu einem Rückruf instruiert. Eine solche Unterstützung würde bei den gemessenen Fehlerquoten die Zahl der erfolgreichen Clientverbindungen um ca. 10% erhöhen.

In einer weiterführenden Analyse könnte man ohne zusätzlichen Programmieraufwand ebenfalls anhand des Clientanteils, der eine Dateieinsicht gestattet, Server ausmachen, bei denen Clients weniger Wert auf Anonymität legen. Auch über Länder, die man von den IP-Adressen der Clients ableiten könnte, ließen sich Aussagen treffen. Das Programm bietet zudem das Gerüst für Inhaltsanalysen, die z.B. anhand der angebotenen Dateien die Interessen von Clients mit bestimmten Merkmalen untersuchen. Wie beim Server Clustering zeigt sich der Ähnlichkeitsbegriff bei Clients wieder flexibel. Der Anteil der verschiedenen Dateien ist nur ein mögliches Kriterium für eine Distanzfunktion. Weitere Eigenschaften wie z.B. Standorte, Antwortzeit oder die Länge der Warteschlangen seiner Dateien sind denkbar.

Für das Berechnen des Clustering wurde hier aufgrund der Inkompatibilität der k-means-Methode für ein hierarchisches Clusteringverfahren entschieden. Auch hier bleibt wieder Raum, die Ergebnisclusterings in Hinblick auf Qualität und Performance anhand diverser Clusteringverfahren zu vergleichen. Ebenfalls bei der Definition der Distanz zu einem Cluster bleibt zu zeigen, ob die „Complete-Link“-Variante oder die „Average-Link“-Methode nicht ein qualitativ besseres Clustering erzeugen.

7 Zusammenfassung

Die aktuell verbreiteten P2P-Netzwerke basieren auf unterschiedlichen Architekturen, die sich in punkto Zentralität stark voneinander unterscheiden. Das eDonkey/-eMule-Netzwerk als ein Modell, in dem Server ein zen-

trales Dateiverzeichnis ihrer Clients pflegen, basiert auf einem vergleichsweise umfangreichen Protokoll, mit dem Clients und Server miteineinander kommunizieren. Für den Handshake beim Verbindungsaufbau werden dabei Nachrichten mit lokalen Informationen ausgetauscht. Die Implementierung dieser initialen Kommunikation, insbesondere einem Thread, der permanent eingehende Nachrichten abfängt und entsprechend interpretiert, bildet das Grundgerüst des Programms, welches im Rahmen dieser Arbeit entwickelt wurde.

Im Prozess der Knowledge Discovery in Databases wurde das Netzwerk auf Muster in der Serverstruktur untersucht, indem ähnliche Server anhand ihrer Größe zusammengefasst wurden. Die Größe eines Servers definierten wir dabei durch die Anzahl der verbundenen Clients sowie die Anzahl der ihm bekannten Dateien. Um möglichst viele Server zu identifizieren, wurden Server nach weiteren ihnen bekannten Serveradressen abgefragt.

Für die Distanzfunktion beim Clustering der Server eignet sich die euklidische Distanz aufgrund der metrischen Skalenniveaus der Variablen. Als Clusteringmethode wurde der k-means-Algorithmus implementiert, der auf der Konstruktion von virtuellen Clustermittelpunkten basiert. Ein Maß für die optimale Anzahl Cluster stellt dabei der Silhouetten-Koeffizient dar, der anhand der durchschnittlichen Klassifizierung aller Objekte die Qualität eines Clusterings ausdrückt.

Bei einem repräsentativen Clustering zeigte sich, dass das eDonkey/eMule-Netzwerk gut hundert „gängige“ Server aufweist, die bei einer optimalen Zerlegung in zwei Cluster aufgeteilt werden. Zwei große Server, die zusammen beinahe ein Viertel aller eDonkey/eMule-Clients bedienten, waren dabei ca. 33 Mal so groß wie 80% der erfassten Server.

Zum Auffinden von ähnlichen Clients machten wir von einer Funktion Gebrauch, welche Einsicht in die Liste der Dateien eines Clients gewährt. Es wurde dazu eine Dateisuchanfrage nach möglichst verbreiteten Dateien an den Server gesendet um möglichst viele Client-IPs zu sammeln.

Wir definierten Ähnlichkeit zwischen Clients über den Anteil der verschiedenen Dateien in ihren Dateilisten. Als Clusteringmethode ist k-means ungeeignet, weshalb ein hierarchisches Clusteringverfahren implementiert wurde, das eine sukzessive Agglomeration der Datenmenge hierarchisch darstellt. Der Algorithmus erlaubt es, Gruppen von Clients mit ähnlichen Interessen zu identifizieren. Der Silhouetten-Koeffizient ist jedoch selbst bei vielen einbezogenen Clients und optimaler Clusteranzahl nur sehr gering, da im Gesamtbild nur wenig Ähnlichkeit zwischen den Clients besteht. Das Clustering zeigt also lediglich eine schwache Struktur, in der die meisten Cluster sich intern nur durch tendenzielle Ähnlichkeit auszeichnen.

Literatur

[BBPW03] BACKHAUS, Klaus ; BERND, Erichson ; PLINKE, Wulff ; WEIBER, Rolf: *Multivariate Analysemethoden*. Springer, 2003

[ES00] ESTER, Martin ; SANDER, Jörg: *Knowledge Discovery in Databases*. Springer, 1999, 2000

[HB02] HECKMANN, Oliver ; BOCK, Axel: The eDonkey 2000 Protocol / Darmstadt University of Technology. Department of Electrical Engineering and Information Technology and Department of Computer Science, Dezember 2002 (8). – Forschungsbericht. Online-Version: ftp://ftp.kom.e-technik.tu-darmstadt.de/pub/papers/HB02-1-paper.pdf

[KB05] KULBAK, Yoram ; BICKSON, Danny: The eMule Protocol Specification / The Hebrew University of Jerusalem. School of Computer Science and Engineering, Januar 2005. – Forschungsbericht. Online-Version: http://www.cs.huji.ac.il/labs/danss/presentations/emule.pdf

[Kli04] KLIMKIN, Alexey: *Unofficial eDonkey Protocol Specification v.0.6.2*. http://prdownloads.sourceforge.net/pdonkey/eDonkey-protocol-0.6.2?download, April 2004

[KM02] KLINGBERG, T. ; MANFREDI, R.: *Gnutella 0.6*. http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html, Juni 2002

[KR05] KUROSE, James F. ; ROSS, Keith W.: *Computer networking*. Addison Wesley, 2005

[Par04] PARKER, Andrew: *The True Picture of Peer-to-Peer Filesharing*. http://www.cachelogic.com/research/slide1.php, 2004

[TNWG92] THE-NETWORK-WORKING-GROUP: *Request for Comments: 1320, The MD4 Message-Digest Algorithm*. http://www.ietf.org/rfc/rfc1320.txt, April 1992

[TNWG02] THE-NETWORK-WORKING-GROUP: *Request for Comments: 3330, Special-Use IPv4 Addresses*. http://www.rfc-editor.org/rfc/rfc3330.txt, September 2002

[Ull04] ULLENBOOM, Christian: *Java ist auch eine Insel*. Galileo Computing, 2004

Abbildungsverzeichnis

Tabellenverzeichnis