



**UNIVERSITÄT PADERBORN**  
*Die Universität der Informationsgesellschaft*

HEINZ NIXDORF INSTITUT  
Algorithmen und Komplexität

## Diplomarbeit

# PeerNear - Ein Netzwerk zur Optimierung der Lokalität in Peer-to-Peer-Netzwerken

Bearbeiter: Markus Scherschanski  
Matrikelnummer: 3269013  
Betreuer: Dr. Christian Schindelhauer  
Bearbeitungszeit: 01.03.2005 - 1.7.2005

## Abstract

Lokalität gewinnt bei dem Design von Peer-to-Peer-Netzen immer mehr an Bedeutung und kann meistens bei schon existierenden Netzwerken nachgerüstet werden, was einen erheblichen Performanzgewinn mit sich bringen kann.

Diese Diplomarbeit stellt ein System namens PeerNear vor, welches durch den Aufruf weniger Traceroutes den Grad der netztopologischen Nähe unter den Peers bestimmen und diese Information jeder Implementation eines Peer-to-Peer-Netzwerks zur Verfügung stellen kann. Prototypisch wurde es als Client-Server-System umgesetzt, es soll jedoch für zukünftige Anwendungen selbst zum Peer-to-Peer-Netzwerk werden.

Bei PeerNear wurde ein neuartiger Algorithmus zur Distanzeinschätzung benutzt, welcher mithilfe der für diesen Zweck eigens entstandenen Zelttheorie, den Verbindungsgraphen zwischen den einzelnen Peers bewerten und neue sinnvolle Erforschungsschritte mittels Traceroute-Aufrufen veranlassen kann. Durch die Clusterung von Knoten zu einem festen Radius, wurde eine wesentliche Verbesserung erzielt. Vor diesem Schritt werden die einzelnen IP-Adressen jedoch schon den Netzen nach zusammengefasst, in denen sie sich befinden. Dazu wurde eine etwas feinere Auflösung als die der Autonomen Systeme gewählt. Mittels einer rekursiven Nutzung des Whois-Dienstes wurden IP-Adressen auf die IP-Bereiche der registrierten Netze, die bei den Regionalen Internet Registraren (RIRs) verzeichnet sind, zugeordnet und indiziert. Damit stellen z.B. sämtliche IP-Adressen der Universität Paderborn nur noch einen Knoten im Netzwerk dar.

Der Algorithmus wurde auf dem Routing-Graphen des Internets getestet. Dabei konnte die Pareto-Eigenschaft des Internetgraphen erneut bestätigt werden, wobei Überlagerungen von verschiedenen Pareto-Verteilungen innerhalb des Graphen festgestellt wurden.

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>iv</b>
<b>1 Einleitung</b>	<b>1</b>
<b>2 Lokalität in Peer-to-Peer-Netzen</b>	<b>4</b>
2.1 Netzwerklokalität . . . . .	4
2.2 Geographische Lokalität . . . . .	5
2.3 Informationslokalität . . . . .	5
2.4 Interessenlokalität . . . . .	6
2.5 Lokalitätsinteraktionen . . . . .	7
2.6 Das Small-World-Phänomen . . . . .	7
2.7 Netztopologische Lokalität . . . . .	8
2.8 Lokalität vs. Repliken . . . . .	9
<b>3 Struktur des Internets</b>	<b>12</b>
3.1 Routing-Protokolle . . . . .	13
3.2 Traceroute . . . . .	14
3.3 Gruppieren von IP-Adressen . . . . .	18
3.4 Erweitertes Traceroute . . . . .	21
3.5 Internet-Graph Eigenschaften . . . . .	22
<b>4 Planen von Traceroute-Aufrufen</b>	<b>26</b>
4.1 Erlernen des Graphen . . . . .	26
4.2 Einführung des Begriffs Zelt als Erweiterung des Graphbegriffs . . . . .	27
4.3 Traceroute im Sinne der Zelttheorie . . . . .	28
4.4 Traceroute Scheduler Algorithmus . . . . .	28
4.5 Erweiterung des Traceroute Scheduler Algorithmus . . . . .	33
4.6 Performanz des Tracescheduler . . . . .	35
4.7 Aufwandsabschätzungen für die A- und B-Berechnung . . . . .	38
<b>5 Das PeerNear-System</b>	<b>40</b>
5.1 Server . . . . .	40
5.2 Clients . . . . .	41
5.3 Peers . . . . .	41
5.4 Benötigte Software und Inbetriebnahme . . . . .	41
5.5 Die Webseite . . . . .	42
5.6 Das Java-Test-Applet . . . . .	43

<b>6</b>	<b>Implementation</b>	<b>45</b>
6.1	Servermodul . . . . .	45
6.2	Clientmodul . . . . .	51
6.3	Java-Peer-Applet . . . . .	51
<b>7</b>	<b>Zusammenfassung</b>	<b>53</b>
<b>8</b>	<b>Ausblick</b>	<b>54</b>
8.1	Verbessertes Traceroute . . . . .	54
8.2	Dynamischer Graph . . . . .	54
8.3	PeerNear als Peer-to-Peer-Netzwerk . . . . .	55
8.4	PeerNear als Routing Protokoll . . . . .	56
	<b>Literaturverzeichnis</b>	<b>I</b>
	<b>RFCs (Requests for Comments)</b>	<b>IV</b>
	<b>Anhang</b>	<b>VI</b>
<b>A</b>	<b>Whois-Eintrag der Uni-Paderborn</b>	<b>VI</b>
<b>B</b>	<b>PeerNear-Graph (vom 6.6.2005)</b>	<b>VII</b>
<b>C</b>	<b>Protokolle</b>	<b>VIII</b>
C.1	Peer über Client zu Server . . . . .	VIII
C.2	Server über Client zu Peer . . . . .	IX
<b>D</b>	<b>Datenbank-Tabellen</b>	<b>X</b>
<b>E</b>	<b>Whois-Cache-Daten (vom 6.6.2005)</b>	<b>XII</b>
<b>F</b>	<b>Dateistruktur</b>	<b>XV</b>
<b>G</b>	<b>Konfigurationdateien</b>	<b>XVI</b>
G.1	Server/config.py . . . . .	XVI
G.2	Server/db/config.py . . . . .	XVI
G.3	Client/config.py . . . . .	XVI
	<b>Ehrenwörtliche Erklärung</b>	<b>XVII</b>

# Abbildungsverzeichnis

1	Peer-to-Peer-Filesharing-Traffic Juni 2004 <i>Quelle: [Par04]</i> . . . . .	1
2	Konzeptüberblick des Emule-Webcache, <i>Quelle: [Koe04]</i> . . . . .	10
3	Traceroute-Aufruf . . . . .	15
4	Problemhaftes Traceroute . . . . .	16
5	Ausschnitt aus der RIPE-Information für das Netz der Universität Paderborn . . . . .	20
6	Ausgabe des erweiterten Traceroute mit Visualisierung . . . . .	21
7	Ergebnisse der Analyse des Wachstums nach Radius . . . . .	23
8	Verteilung der Knotengrade . . . . .	24
9	Überlappen der Pareto-Verteilungen . . . . .	25
10	Elemente der Zelttheorie . . . . .	28
11	Traceroute als fixiertes Seil an einer Stange . . . . .	28
12	Einfügen einer Stange in die B-Matrix . . . . .	31
13	Beispiel mit zwei Repräsentanten und Radius 2 . . . . .	34
14	Zugehörige Matrizen . . . . .	34
15	Traceroutes pro Netze . . . . .	35
16	Repräsentanten pro Netze . . . . .	36
17	Knoten pro Repräsentanten . . . . .	36
18	Durchschnittliche Überschneidung von Repräsentanten . . . . .	37
19	Anzahl der Überschneidungen . . . . .	38
20	Peer-Client-Server-Struktur . . . . .	40
21	Webseite mit gestartetem Applet . . . . .	42
22	Übersichtsdiagramm des Servermoduls . . . . .	45
23	Klassendiagramm Graph-Algorithmen . . . . .	49
24	Klassendiagramm Client-Modul . . . . .	51
25	Klassendiagramm Java-Peer-Applet . . . . .	52

# 1 Einleitung

Peer-to-Peer-Netzwerke erfreuen sich immer größerer Beliebtheit, besonders die Ausprägung, die man als Filesharing-Netzwerke bezeichnet, wie z.B. eDonkey, BitTorrent, Kazaa, Gnutella, usw., um nur die bekanntesten zu nennen. In diesen Netzen geht es darum, große Dateien im Internet möglichst effizient verbreiten zu können, ohne daß alle die Datei bei einem einzigen Server herunterladen müssen. Man setzt dabei auf die epidemische Ausbreitung von den einzelnen Teilen der Dateien. Filesharing wird mittlerweile als Synonym für Peer-to-Peer-Netzwerke gebraucht. Allerdings ist ein Peer-to-Peer-Netzwerk lediglich ein Kommunikationsmodell in dem jeder Teilnehmer eines Netzwerks die gleichen Aufgaben übernimmt und somit sozusagen als gleichberechtigt fungiert bzw. auch gleichgestellt ist, wobei das Netzwerk welches hierbei gemeint ist, nicht unbedingt nur ein Rechnernetzwerk sein muß, sondern dieser Begriff auch für soziale Netzwerke gelten kann. Laut dem letzten *Global Internet Geography-Reports* soll Filesharing bereits 24 Prozent des gesamten Internet-Verkehrs ausmachen [Pri04]. CacheLogic [Par04] sieht die Lage noch dramatischer: in Europa soll der Anteil der Filesharing-Nutzung schon über 50 Prozent des gesamten Internetverkehrs (=Traffic) ausmachen (siehe Abbildung 1). Tatsächlich könnte dieser Anteil noch sehr viel höher liegen, da der „Peer-to-Peer-Traffic“ durch neuere, individuellere Benutzung von Port-Adressen nicht mehr eindeutig identifizierbar ist [TK04].

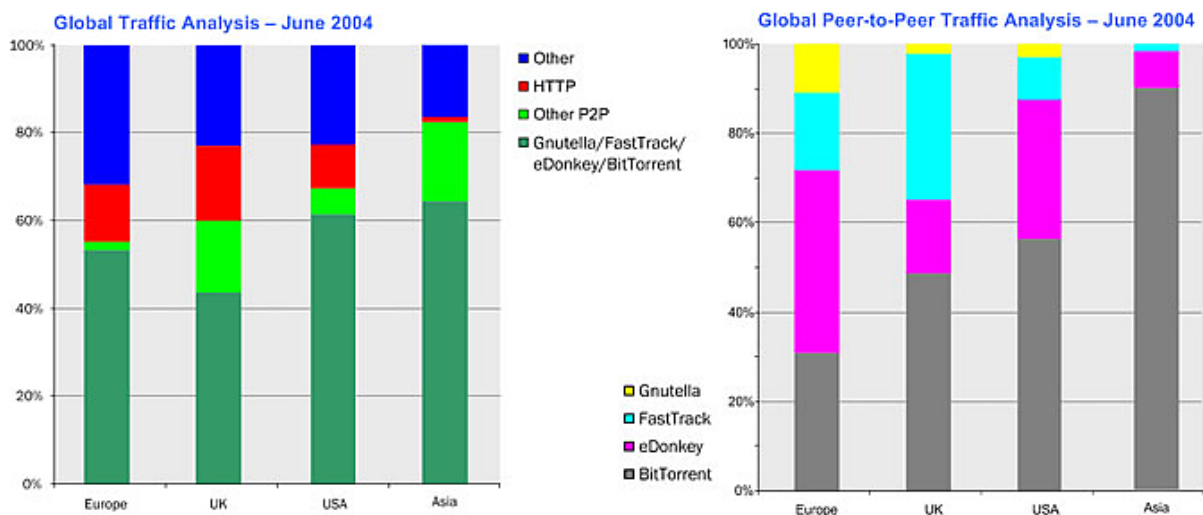


Abb. 1: Peer-to-Peer-Filesharing-Traffic Juni 2004 *Quelle: [Par04]*

Mittlerweile wird dieser Sachverhalt als Problem, fast sogar als Bedrohung für den ordnungsgemäßen Betrieb des Internets angesehen, zumal die Anzahl der Filesharing-Nutzer immer weiter steigt. Laut BayTSP [Bor04] hatten im September 2004 eDonkey und Kazaa zusammen 5 Millionen Benutzer täglich, wobei das beliebteste Protokoll BitTorrent noch nicht berücksich-

tigt wurde. Es gibt Schätzungen, die bis zu 30 Millionen verschiedene Filesharing-IP-Adressen und somit User weltweit angeben [Par04]. Der Druck ist also hoch um Traffic zu vermeiden, wo es nur geht.

Am meisten kosten die Filesharing-Nutzer den einzelnen Provider, der für jedes Gigabyte Traffic zahlen muss. Auch wenn die Preise pro Gigabyte immer weiter nach unten gehen, können ein paar Filesharing-User einen Provider in den Ruin treiben. Erst vor kurzem hat sogar ein großer Deutscher Provider, seinen „Powerusern“, die mehr als 20 GB/Monat durch die Leitung übertragen und somit sehr wahrscheinlich Nutzer von Filesharing-Tauschbörsen sind, 100 Euro angeboten, wenn sie den Flatrate-Provider wechseln [Sch05]. Provider würden sehr viel Geld sparen, wenn sich der Filesharing-Traffic fast ausschließlich innerhalb ihrer eigenen Strukturen bewegen würde, weil sie für diese nicht pro Gigabyte, sondern nur für die laufenden Kosten aufkommen müssten. Außerdem könnten sie auch dort die Bandbreiten beliebig erweitern, da eine externe Verbindung zu verbreitern sehr viel mehr kostet, als das Erweitern einer internen. Selbst wenn der Verkehr auch mal zu externen Knoten gehen müsste, wäre es doch viel effizienter, einen Knoten auszuwählen, der nicht weit von dem eigenen Provider entfernt ist, da dadurch in der Summe auch weniger Leitungen belastet würden. Anstattdessen sind für heutige Peer-to-Peer-Netzwerke sehr weit entfernte Verbindungen (z.B. in die USA) genauso gut, wie eine Verbindung zu einem Peer bei dem gleichen Provider und verstopfen somit die Überseeleitungen.

Die Peer-to-Peer-Netze müssen also lokaler werden! Doch wie kann man Peer-to-Peer-Netze dazu bringen, lieber nahe Verbindungen zu wählen, als weit entfernte, bzw. wie sollen die Peers überhaupt herausfinden, wie weit eine gewisse IP-Adresse von ihnen entfernt ist? Wie kann man bei den existierenden Peer-to-Peer-Netzwerken dieses Lokalitätsbewußtsein nachrüsten? Bringt Lokalität noch andere Vorteile? Gibt es überhaupt etwas anderes als Filesharing?

Dieses kann durch die Erfahrungen mit den wenigen lokalitätsbezogenen, nicht filesharing-orientierten Peer-to-Peer-Netzen, wie dem Plethora-Peer-to-Peer-Netzwerk [FJG04] bejaht werden. Es hatte bewiesen, daß Netzlokazität die Latenzzeiten zwischen den Peers um bis zu 60 Prozent verbessern kann. Deshalb scheint es unerläßlich eine Möglichkeit zu schaffen, welche diese Strukturkenntnis bei allen bisherigen Peer-to-Peer-Netzwerken und Programmiersprachen systemunabhängig nachrüsten kann, bzw. ein System zu schaffen, auf dem neue Peer-to-Peer-Ansätze schon in der Planungsphase direkt basieren könnten.

Als Alternative dazu, könnte man zwar auch Multicast- und Broadcastmechanismen auf IP-Basis einsetzen, wobei dann die Router dafür zuständig wären, selbständig die beste Route zu suchen. Nur leider sind diese in ihrer Entwicklung stagniert und finden faktisch keine Anwendung, da diese Protokolle von den meisten Routern nicht unterstützt werden [HHX<sup>+</sup>03]. Die Lösung um trotzdem Multicast betreiben zu können, sind ALM (Application Layer Multicast)-Technologien, die anstatt auf den tieferen Schichten des Internet Protokolls, auf der Anwender-

schicht arbeiten, d.h. sie nehmen die Struktur des Internets als gegeben hin und etablieren darauf eine eigene Netzstruktur. Somit sind diese auch darauf angewiesen die Netzstruktur zwischen den Peers zu kennen.

Um zu wissen, wie nahe sich zwei Peers im Internet stehen, müsste man also wissen, wieviel Distanz zwischen ihnen liegt. Doch was ist Distanz im Internet? Die Anzahl der Router zwischen beiden? Die Geschwindigkeit mit der die beiden Daten austauschen können? Die Geschwindigkeit, mit der einer auf den anderen reagieren kann (Latenzzeit)? Was ist am sinnvollsten? Egal welche Möglichkeit man wählt, um die perfekten Knotenpaarungen zu erfahren, müsste jeder Peer mit allen anderen Peers einen Test durchführen, um die beste Paarung zu erhalten. Wenn man jedoch die Wege, die die Pakete zwischen den Peers nehmen, mitprotokolieren könnte und aus diesen Informationen einen Graph zeichnen würde, könnte man schon nach einigen Kommunikationen „orakeln“, welche Teilnehmer näher beinander stehen und welche weiter weg voneinander sind, ohne sofort alle Tests durchführen zu müssen, und je mehr Aufrufe man tätigt, desto genauer kann man die Nähe, bzw. die Lokalität vorhersagen. Das Hilfsmittel, um solche Pfade durchs Internet aufzeichnen zu können, nennt sich Traceroute und es gibt einige Projekte, z.B. Skitter [HPMc02], IDMAPS [PF01] oder NetDimes [Sha04], die versuchen mithilfe dieses Tools die Struktur des Internets zu erforschen. Die Methode ist dabei stets gleich: sogenannte „Tracer“ führen Traceroutes zu (meist zufällig ausgewählten) IP-Adressen von verschiedenen Standorten auf der Welt her aus. Diese Informationen werden zentral zu einem Graph zusammengeführt.

Dieser Ansatz wird in dieser Diplomarbeit weitergeführt, es ist jedoch nicht das Ziel, den kompletten Internetgraph erforschen, sondern nur den Teil, der zwischen den Peers in einem Peer-to-Peer-Netzwerk liegt und das mit möglichst wenig Aufrufen des Traceroute-Programms. Danach soll den Peers die Möglichkeit gegeben werden, sich zu jedem Zeitpunkt informieren zu können, welche anderen Peers aus dem gleichen Netzwerk am nächsten zu ihnen selber stehen. Das ist die Intention von PeerNear, dem Resultat dieser Diplomarbeit.

Um das System von PeerNear näher zu bringen, soll in Kapitel 2 der Begriff der Lokalität näher definiert und analysiert werden. In Kapitel 3 sind danach die Struktureigenschaften des Internetgraphs und Möglichkeiten zu dessen Erforschung Thema. In Kapitel 4 wird der Algorithmus vorgestellt, der mit möglichst wenigen Traceroute-Aufrufen die Netzstruktur zwischen den Peers erforschen kann, sozusagen das Kernstück von PeerNear. Zum Schluss führt sich alles zu dem Gesamtkonzept von PeerNear zusammen und die besonderen Feinheiten der Implementation werden vorgestellt.

## 2 Lokalität in Peer-to-Peer-Netzen

Lokalität in Peer-to-Peer-Netzen ist als räumliche Nähe von Peers zu verstehen, wobei der Begriff Raum verschiedene Bedeutungen haben kann, je nach dem Gebiet, auf den er sich bezieht. Es gibt geologische, mathematische, logische oder auch soziale Räume. Eine wesentliche Eigenschaft von Räumen ist, daß sie teilbar sind. Das Internet kann man auch als Raum verstehen, den man nach verschiedenen Kriterien einteilen kann. Zum Beispiel kann man die am Internet teilnehmenden Rechner nach ihrer geologischen Position und Nähe gruppieren (z.B. alle Rechner in Deutschland). Kleine Gruppen von Rechner die nicht weiter als ungefähr einen Kilometer auseinanderliegen und untereinander vernetzt sind, faßt man als LAN (Local Area Network) zusammen, worin auch wieder das Wort lokal vorkommt. Dies demonstriert übrigens sehr gut, daß man das Internet als ein großes Netzwerk sehen kann, oder als ein Netzwerk von Netzwerken, wobei Netzwerk nicht nur auf die physikalische Verbindung von Rechnern zu beziehen ist. Soziale Netzwerke existieren im Internet genauso und können ebenfalls lokal begrenzt sein. Zum Beispiel wird man auf amerikanischen Webservern viel weniger Informationen über den aktuellsten deutschsprachigen Popsong finden, als in deutschsprachigen Gefilden. Die Informationen sind also lokal begrenzt.

Das Internet kann man somit in verschiedene Arten von Räumen aufteilen. Was jedoch ist das Internet? Im Prinzip sind es nur Verbindungen zwischen Rechnern und diese Verbindungen haben eine eigene Struktur. Alle Anwendungen die über das Internet kommunizieren, überlagern diese Struktur in irgendeiner Form, sogar der einzelne Rechner, der sich eine Webseite herunterlädt, erzeugt eine überlagernde Kante auf dem Internet-Graph. Das heißt wir müssen das Internet in einen Verbindungsgraphen und eine unendliche Menge von „Overlay-Graphen“, die sich je nach Anwendung voneinander unterscheiden, unterteilen.

Wenn die Anwendungen, die auf dem Internet laufen sollen, Peer-to-Peer-Netzwerke sind, dann muss sich deren Struktur in die Struktur des Internets einpassen, um gut funktionieren zu können. Bei manchen ist dies eine Ringstruktur, bei manchen eine Sternform, bei manchen auch eine Struktur, die keine feste Form hat. Also was kann man verbessern, ohne auf die einzelnen Strukturen eingehen zu müssen? Man versucht die Verbindungen zwischen den einzelnen Knoten so kurz wie es geht zu halten. Dieses nennt man Netzwerklokalität.

### 2.1 Netzwerklokalität

Die Netzwerklokalität ist das Gütemaß, wie nah zwei Knoten eines Overlay-Graphen, gemessen im Internetgraphen, tatsächlich sind.

Diese Nähe kann man durch verschiedene Methoden messen:

**RTT (Round Trip Time)** Dabei wird die Zeit gemessen, die eine Anfrage zu einem anderem

Peer und wieder zurück benötigt. Meistens wird dabei auch ein Mittelwert über mehrere Messungen gebildet. Es wird ein Nachrichtenpaket (ping) von Peer  $p_1$  nach  $p_2$  versendet und ein Zeitgeber gestartet  $t_1$ . Wenn  $p_2$  das Paket erhält, schickt er es sofort postwendend zurück (pong). Wenn das Paket bei  $p_1$  wieder angekommen ist, wird der Zeitgeber wieder gestoppt  $t_2$ . Die gemessene Zeit  $RTT = t_2 - t_1$  ist die Round Trip Time. Bei diesem Verfahren wird angenommen, daß die RTT proportional zur Entfernung zweier Knoten ist.

**Hop-Distanz** Dabei wird berücksichtigt, wieviele Knoten zwischen den Peers existieren. Dazu wird ein Datenpaket generiert, welches nach  $n$  Knoten eine Rückmeldung des betreffenden Knoten auslöst. Also angenommen  $p_1$  schickt ein Paket mit dem Ziel  $p_2$  und setzt die Lebenszeit (Time-To-Live=TTL) des Pakets auf  $n$ , dann wird der  $n$ -te Knoten auf dem Weg zwischen  $p_1$  und  $p_2$  eine Nachricht an  $p_1$  schicken, sobald das Paket bei ihm eingangen ist. Das wird für  $n=1 \dots maxhops$  wiederholt. Danach weiß man, welche Knoten auf dem Weg zwischen  $p_1$  und  $p_2$  liegen.

Die gemessenen Netzwerkeigenschaften zwischen zwei Knoten nennt man Metrik.

## 2.2 Geographische Lokalität

Zwei Peers sind dann geographisch lokal, wenn Sie sich z.B. auf dem gleichen Kontinent, in dem gleichen Land oder der gleichen Stadt befinden.

Man kann diesen Begriff auch erweitern, da Kontinente oder Länder auch nichts anderes als räumliche Begrenzungen sind. Es können innerhalb von Räumen auch virtuelle Begrenzungen geschaffen werden, z.B. durch einen mathematischen Gruppierungsalgorithmus. Dies macht z.B. GeoPeer [AR04], welches ein Peer-to-Peer-Netzwerk ist, das die Knoten mathematisch durch eine Delaunay Triangulation in Bereiche einteilt und dadurch eine virtuelle geographische Lokalität erreicht.

## 2.3 Informationslokalität

Informationslokalität ist dann gegeben, wenn die Entfernung des Speicherorts zweier Informationen proportional zur Ähnlichkeit der Informationen abnimmt.

Dabei kann sich die Informationsdistanz über mehrere Peers erstrecken, aber auch auf dem gleichen Peer weitergeführt werden. Sehr gut verinnerlicht dieses Konzept Skipnet [HJS<sup>+</sup>03]. Skipnet ist ein Peer-to-Peer-Netzwerk, bei dem eine Abbildungsmenge (z.B. Buchstaben) äquidistant auf einer Ringstruktur verteilt ist, wobei jeweils ein Knoten auf dem Ring ein gewisses

Prefix darstellt, abhängig von der Verschachtelungstiefe. Durch einen Algorithmus, der Skiplisten sehr ähnelt, werden Suchanfragen auf dem Ring herumgeschickt, wobei der Besuch der einzelnen Knoten einen eindeutigen binären Pfad hinterläßt. Dieser Pfad, sowie der Speicherort der Daten, sind informationslokal.

Informationslokalität kann man auch durch Manipulationen an existierenden Peer-to-Peer-Netzwerken erreichen, z.B. indem man die Hashfunktion, die man bei den meisten Peer-to-Peer-Netzen benötigt, verändert. Allerdings handelt man sich damit auch viel Unflexibilität und Datenhäufungen ein, die eher unerwünscht sind. Außerdem ist es wirklich schwer eine vernünftige Informationslokalität bei existierenden Peer-to-Peer-Netzwerken nachzurüsten.

Beispiele für Eigenschaften die Informationslokalität erzeugen können:

- Schlüssel die mit A beginnen, werden auf dem gleichen Knoten gespeichert.
- Schlüssel die nur einen Anfangsbuchstaben auseinander liegen, dürfen auch nur maximal einen Knoten voneinander entfernt sein.

## 2.4 Interessenlokalität

Interessenlokalität ist dann gegeben, wenn Peers, die die gleichen Ziele verfolgen, auch im Netzwerk nahe beieinander liegen.

Als Beispiel hat Sripanidkulchai [SMZ03] eine Erweiterung für das Gnutella-Netzwerk vorgeschlagen, die dieses um den Faktor der Interessenlokalität erweitert. Dabei beobachtet der einzelne Peer die Suchanfragen, die durch ihn hindurchgehen und merkt sich die Peers, die ähnliche Suchanfragen stellen, wie der Peer selber. Falls eine hohe Ähnlichkeit zwischen den Suchanfragen besteht, baut der Peer eine direkte Verbindung (Shortcut) zu dem Peer mit der hohen Ähnlichkeit auf. Wenn also der Peer das nächste Mal selber etwas sucht, ist somit die Wahrscheinlichkeit viel höher, daß er schnell fündig wird, da der Peer, der die selben Interessen hat wie er, wahrscheinlich auch im Besitz der gesuchten Daten ist.

Interessenlokalität kann man also durch das Design des Peer-to-Peer-Netzwerks erlangen, wenn dieses es auch zulässt. An dem Beispiel von Gnutella sieht man, daß dies möglich ist. Allerdings würde es schwierig bei Netzen, die einen konstanten Knotengrad bzw. eine feste Struktur haben und nicht jede Suchanfrage an alle Peers senden. Ein Beispiel für eine Eigenschaft, welche Interessenlokalität bezüglich „Heavy Metal“ herstellt:

- Peers die viele „Heavy Metal“-Musikdateien besitzen, liegen untereinander näher zusammen als zu Peers, die eher HipHop-Musik favorisieren.

## 2.5 Lokalitätsinteraktionen

Nun kann jedoch keine Lokalität alleine für sich sehen, da sich alle Lokalitäten gegenseitig bedingen können. Zum Beispiel möchten Peer-to-Peer-Nutzer, die Kunden bei der Deutschen Telekom sind, auch meist nur deutschsprachige Inhalte haben. Sie sind demnach durch die netzwerktechnische Nähe (Netzwerklokalität) verbunden und durch ihr Interesse nach deutschsprachigen Inhalten (Interessenlokalität). Außerdem wohnen Sie alle in Deutschland (Geographische Lokalität). Man wird umgekehrt aber auch die meisten deutschen Inhalte im Netz bei der Deutschen Telekom finden, da diese auch der größte deutsche Provider ist und somit ist ebenfalls die Informationslokalität gegeben. Dieses Phänomen wurde schon von anderen beobachtet und wird als „Small-World-Phänomen“ bezeichnet.

## 2.6 Das Small-World-Phänomen

Das Small-World-Phänomen wurde von Sozialpsychologen in den 60er Jahren aufgebracht und beschreibt, daß jeder Mensch auf Erden mit jedem anderen Mensch über eine überraschend kurze Kette von Kontakten verbunden ist.

Hop-Distanzen im Internet sind meistens auch nicht sehr lang, was auch so gewollt ist. Jeder Rechner kann jeden anderen Rechner im Internet mit einer kleinen Anzahl von Zwischenvermittlern (Routern) erreichen.

Nach Watts [Wat99] zeichnen sich Small-World-Graphen durch folgende Eigenschaften aus:

**Geringer Durchmesser** Der Durchmesser  $d$  in solchen Graphen ist ziemlich gering und wächst nur sehr langsam mit zunehmender Knotenanzahl  $n$ , z.B.  $d = O(\log n)$ .

**Geringe Dichte** Die Knoten von solchen Graphen sind nur sehr spärlich untereinander verbunden. Die Anzahl der Kanten im Verhältnis zu den Knoten  $n$  ist z.B. nur  $O(n)$ .

**Clusterbildung** Die Wahrscheinlichkeit ist sehr hoch, daß wenn zwei Knoten eine Kante unterhalten, große Teile ihrer Nachbarschaft sich ebenso ähneln.

Das bedeutet: wenn der Internet-Graph die Eigenschaften eines Small-World-Graphen besäße und man Lokalität auf der Ebene der Netzwerktopologie herstellen würde, müssten die anderen Lokalitäten auf dem Fuß folgen. Die Frage ist nur, ob der Internet-Routing-Graph diese Veranlagungen mitbringt. Dies wird sich in Kapitel 3.5 dieser Arbeit klären, jedoch um zu wissen, wie man netztopologische Lokalität genau herstellt, muß man sich vorher klar machen, was das eigentlich bedeutet.

## 2.7 Netztopologische Lokalität

Der große Unterschied zwischen der netztopologischen Lokalität zur Netzwerklokalität besteht darin, daß z.B. durch die RTT-Messung nicht unbedingt die Knoten näher beieinander gesehen werden, die auch die wenigste Anzahl an physikalischen Verbindungen unterhalten. Gleiches gilt für die Hopdistanz. Falls Router falsch konfiguriert sind, können Peers weiter entfernt erscheinen (oder näher), als das sie es tatsächlich sind.

Die einzige Möglichkeit diesem Problem zu begegnen, ist einen kompletten Graph von der Netzstruktur zwischen den Peers zu zeichnen. Als Messinstrument kann man dazu wieder die Hopdistanz-Messung verwenden, jedoch zerteilt man hierzu das Ergebnis in die einzelne Verbindungsinformationen: z.B. wird aus der Ergebnisknotenfolge  $H = \langle 1, 2, 3, 4 \rangle \mapsto \{(1, 2), (2, 3), (3, 4)\}$ , die Liste der Einzelkanten, aus der man dann den Verbindungsgraph des kompletten Netzwerks zeichnen kann.

### 2.7.1 Nachrüsten der Netztopologischen Lokalität

Eine Operation, die für alle bisherigen Peer-to-Peer nachrüstbar wäre, ist:

Wenn eine Verbindung zu einem anderem Peer aufgebaut werden soll (auch wenn nur ein Datenpaket gesendet wird (z.B. UDP)), frage vorher nach, ob es nicht eine Verbindung gibt, die netztopologisch näher liegen würde. Wenn ja, benutze diese, außer es spricht ein triftiger Grund dagegen (z.B. der Algorithmus selber).

Eine Erfolgsgeschichte dazu liefert das sehr bekannt gewordene Chord Peer-to-Peer-Netzwerk [MKKB01]. Dieses besaß im ersten Entwurf keine Ansätze um die Lokalität einzubeziehen, jedoch hat Dabek [DLS<sup>+</sup>04] mit der Erweiterung DHASH++ eine Reihe von Techniken vorgestellt, um Netzwerklokalität nachzurüsten (noch keine netztopologische). Die Haupttechnik dabei nennt sich Proximity Neighbor Selection (PNS), bei der die Routingtabellen hinsichtlich der Roundtrip-Zeiten (RTT) optimiert werden.

### 2.7.2 Vorteile der netztopologischen Lokalität

Ein Problem bei den Roundtrip-Zeiten ist, daß für jede Knotenkombination, der Test erfolgen muss, welcher die Roundtrip-Zeit mißt. Außerdem sind diese Messungen sehr fehleranfällig, da sie durch viele Faktoren, wie z.B. die Auslastung des Zielrechners, variieren können. Ein Zielrechner kann nämlich nur so schnell antworten, wie er das Paket auch verarbeiten kann. Stattdessen sollte die netztopologische Lokalität bei Peer-to-Peer-Netzwerken nachgerüstet und bevorzugt werden. Folgende Vorteile sprechen dafür:

---

**Reduzierung der Tests** Um einen Graphen aus den Pfaden der Hopdistanz-Messung herzustellen, der alle Peers untereinander verbindet, müssen nicht alle Paarungen getestet werden.

**Auch mehr andere Lokalität** Unter der Annahme, daß alle Lokalitätsformen in der Realität dicht verwoben sind, wofür einige Indizien genannt wurden, könnten durch die Herstellung von netztopologischer Lokalität auch alle anderen Lokalitätsformen gleichzeitig verbessert werden.

**Kostenersparnis** Wenn existierende Peer-to-Peer-Netzwerke die netztopologische Lokalität berücksichtigen würden und z.B. nur noch Peers miteinander direkt Daten austauschen würden, die auch sehr nahe in der Netzwerkstruktur stehen, meist vielleicht sogar beim gleichen Provider sind oder sich bestenfalls am gleichen Router befinden, könnten die Kosten für die Provider extrem sinken, da interner Verkehr diese so gut wie nichts kostet.

**Mehr Bandbreite für andere Anwendungen** Ein anderer Nebeneffekt wäre, daß dadurch mehr Kapazität für Nicht-Peer-to-Peer-Applikationen zur Verfügung stünde. Es könnten sogar die Peer-to-Peer-Netze selber einen „Speed-Up“ erfahren, wenn z.B. zuerst die Daten von näher liegenden Knoten (die somit auch mehr Bandbreite haben) geholt würden und dann erst auf weiter entfernte Verbindungen ausweichen.

**Sicherheit und Anonymität** Netztopologische Lokalität bringt sogar mehr Sicherheit mit sich, da sich Angreifer, um komplexe Attacken gegen einzelne Peers fahren zu können (z.B. ihn einzukreisen), in dessen Nähe begeben müssen. Wenn das Netzwerk jedoch auf die netztopologische Lokalität achtet, werden die feindlichen Peers nie dort hingelangen, falls sie sich nicht netztopologische nahe bei dem Ziel aufhalten. So würde z.B. ein feindlicher Peer in Köln extreme Schwierigkeiten haben, an einen Peer in den USA heranzukommen, da der Zielppeer immer Verbindungen in seinem Netz in den USA, demjenigen in Köln vorziehen würde. Feindliche Peers, die andere Peers ausspionieren wollen, haben das gleiche Problem. Sie müssten einen erheblichen Aufwand betreiben, um an den Zielppeer heranzukommen, z.B. in dem Ort, wo sich der Zielppeer aufhält, einen Anschluß bei dem gleichen Provider anmelden.

## 2.8 Lokalität vs. Repliken

Daß der Kostenfaktor bei Peer-to-Peer-Netzwerken nicht mehr wegzudenken ist, zeigt unter anderem die Entwicklung von Peer-to-Peer-Cache-Programmen, wie z.B. PeerCache [Jol05] oder der eMule-Webcache [Koe04]. Diese verfolgen eine andere Strategie: Um Daten dort zu platzieren, wo sie am meisten angefragt werden, muss man sie replizieren. So steht diese Möglichkeit

auch immer wieder gern zur Debatte, wenn es um das Beschleunigen von Peer-to-Peer-Netzen geht. Dies ist jedoch meist die letzte Ausflucht, wenn das Netzwerk-Design keine weiteren Verbesserungsmöglichkeiten zuläßt oder Nadelöhre entstehen.

Auch bei dem berühmten Edonkey-Netzwerk wird diese Möglichkeit seit neuestem genutzt. Dazu werden die HTTP-Proxyserver des eigenen Providers missbraucht, indem Pseudo-HTTP-Anfragen auf andere Emule-Peers durchgeführt werden, die sich ausserhalb des Providers befinden. Wenn diese Daten dann einmal in den Cache des Proxyserver geladen wurden, können beliebig viele andere Peers die Daten direkt aus dem Cache des Proxyserver herunterladen, solange sie die gleiche URL wie der erste Peer dabei angeben. In Abbildung 2 wird das komplette Verfahren visualisiert. Der Vorteil ist natürlich, daß unter anderem auch die Bandbrei-

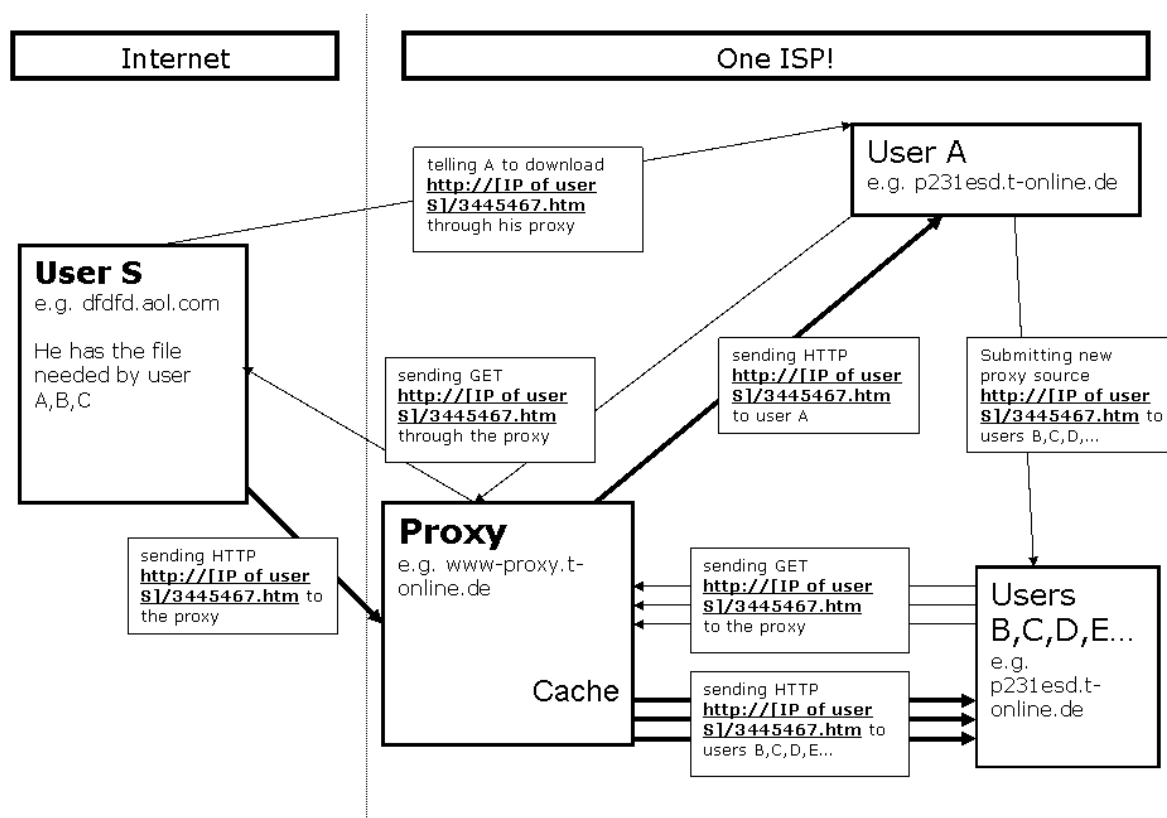


Abb. 2: Konzeptüberblick des Emule-Webcache, *Quelle: [Koe04]*

tenbeschränkungen der Provider umgangen werden und somit, wenn die Daten erst einmal mit kleiner Geschwindigkeit hochgeladen wurden, mit voller Geschwindigkeit an alle anderen Peers verteilt werden können. Jedoch ist das Ganze eher ein „herumtricksen“, welches unheimlich viel Verkehr auf dem Proxyserver des Provider verursacht und wahrscheinlich bald unterbunden wird. Gegen die Lösung per netztopologischer Lokalität ist jedoch nichts zu sagen, wenn

alle Teile innerhalb eines Providers ausgetauscht werden und bei Bedarf neue Datenfragmente von Providerexternen Peers geholt werden. Dies ist auf jeden Fall der Cache-Lösung vorzuziehen, da es kein zentrales Nadelöhr schafft.

### 3 Struktur des Internets

Um die Struktur des Internets erforschen und beschreiben zu können, brauchen wir als erstes Begrifflichkeiten, die die Bestandteile des Internets eindeutig identifizierbar machen:

**IP-Adresse** 32-Bit-Wert, mit dem ein Rechner, der an das Internet angeschlossen wird, eindeutig identifizierbar und somit auch adressierbar ist. Für die bessere Lesbarkeit werden die einzelnen Bytes des Wertes als Dezimalzahlen dargestellt, die durch jeweils einen Punkt getrennt sind, z.B.: 1.2.3.4

**IP-Registare** Lokale, regionale und weltweite Organisationen, die für die Vergabe von Internet-Adressen-Bereichen zuständig sind.

**IP-Paket** Datentransfereinheit im Internet. Als Analogie kann ein Postpaket herangezogen werden. IP-Pakete bestehen aus einem Kopf- (Header) und einem Datenteil. Der Kopfteil wird dazu benutzt, den Datenteil korrekt adressieren zu können.

**Peer** Einzelner Rechner, der netzwerktopologisch einen Endpunkt darstellt.

**Router** Rechner, der ausschließlich IP-Pakete weiterleitet und dabei vermittelt. Netzwerktopologisch ein innerer Knoten.

**Autonomes System (AS)** Netzwerk, welches als eine Einheit verwaltet wird und innere Strukturen hat, sich aber nach außen hin als Ganzes darstellt. Es besteht aus Routern und Peers. Autonome Systemen bekommen eine Identifikationsnummern zugewiesen (ASN), um für Routing-Protokolle zuordbar zu sein.

**Provider** Kommerzieller Anbieter von Internet-Dienstleistungen. Besitzer eigener Netzwerkstrukturen. Er bildet meist ein Autonomes System.

**Backbone** Rückgrat des Internets: sehr große, über eine hohe Bandbreite verfügende Rechner, die meist ringförmig über Glasfaserleitung miteinander verbunden sind.

**Route** Eine Folge von IP-Adressen der Router, die ein Paket von einem Peer zu einem anderem Knotenpunkt besucht. Dabei kann der Zielknoten auch ein Router sein.

**AS Path** Eine Folge von Autonomen Systemen, die auf dem Weg zwischen zwei Autonomen Systemen liegen.

## 3.1 Routing-Protokolle

Die Struktur des Internets wird durch die Verbindungen der einzelnen Router untereinander und der Art und Weise, wie diese Router IP-Pakete vermitteln, vorgegeben. Damit Router wissen, wohin sie welche Pakete weiterleiten sollen, bzw. damit diese Pakete auch bei ihrem Ziel und das schnellstmöglich ankommen, existieren verschiedene Protokolle die diese Inter-Router-Kommunikation regeln.

Welches nun an welcher Stelle des Internets verwendet wird, hängt von den speziellen Eigenschaften der Protokolle ab. Dabei unterscheidet man zwischen Intra-AS- und Inter-AS-Routing-Protokollen. Intra-AS-Protokolle sind nur dafür geeignet, innerhalb von Autonomen Systemen als Routing-Protokoll zu fungieren, im Gegensatz zu Inter-AS-Protokollen, die für das Routing zwischen den Autonomen Systemen geeignet sind.

### 3.1.1 RIP - Routing Information Protocol

Beim RIP-Protokoll [1, 2] sendet der Router am Anfang seine Routingdaten an die umliegenden Nachbarn und fordert dies umgekehrt auch. Daraus berechnet er mittels des Distanzvektor-Algorithmus die Kosten zu jedem anderem Router im Netzwerk. Der Austausch der Daten wird in bestimmten Intervallen wiederholt, um Änderungen im Netzwerk berücksichtigen zu können.

Ein schweres Problem des Distanzvektor-Algorithmus ist es, daß neue Verbindungen zwar schnell erkannt werden, jedoch wegfallende Verbindungen sehr lange brauchen, bis diese sich herumgesprochen haben (Count-to-Infinity-Problem).

RIP ist nur geeignet für sehr kleine Netze, um z.B. eine Default-Route durch das Netzwerk zu etablieren, deshalb wurde dieses Verfahren von OSPF als Intra-AS-Verfahren abgelöst.

### 3.1.2 OSPF - Open Shortest Path First

Das OSPF [3, 4] wird meist innerhalb von Autonomen Systemen verwendet und ist ein Link-State-Protokoll, d.h. ein Router teilt durch „Flooding“ von Status-Nachrichten dem Rest des Netzwerks mit, welche Verbindungen er unterhält. Dadurch kann jeder andere Router eine Adjazenz-Matrix des Netzwerks aufbauen und mithilfe des Shortest-Path-First-Algorithmus, die optimalen Wege bestimmen.

Damit die Routing-Tabellen klein bleiben, wird das Autonome System nochmals in Abschnitte (Areas) unterteilt, welche jeweils eine eigene Nummer haben. Die Areas werden über einen Backbone (Area 0) zusammengehalten. OSPF kann sehr große Autonome Systeme verwalten, ist aber trotzdem nicht leistungsfähig genug für das Inter-AS-Routing.

### 3.1.3 BGP - Border Gateway Protocol

Damit die Autonomen Systeme untereinander geroutet werden können (Inter-AS), wird das BGP-Protokoll [5, 6, 7] eingesetzt.

Zu Beginn tauschen alle Autonomen Systeme ihre kompletten Routingdaten aus, bzw. führen sie zusammen. Diese Information wird jede Minute aktualisiert, allerdings werden dabei nur noch die Veränderung in den Verbindungen übertragen. Zur Berechnung der Routen benutzt BGP den Pfad-Vektor-Algorithmus, d.h. anstatt der Distanzen werden die kompletten Pfade zwischen den Routern berücksichtigt und man vermeidet dadurch das Count-To-Infinity-Problem. Außerdem können bei BGP viele manuelle Eingriffe gemacht werden, was das Routing anbetrifft. Dies ist wichtig, damit Datenverkehr nicht über „feindliches“ Territorium geleitet wird. Es wäre sehr ungünstig, wenn z.B. der Datenverkehr von Microsoft über das Netz von IBM geroutet würde.

In der Vergangenheit wurden viele Versuche unternommen, diese Routinginformationen anderweitig nutzbar zu machen. So könnte man sie auch verwenden, um die netztopologische Lokalität zwischen Peers ermitteln zu können, bzw. zu optimieren. Das Problem ist nur, daß BGP-Informationen von Peers aus nicht abrufbar sind, da sie nur auf den betroffenen Routern existieren, bzw. diese Informationen viel zu umfangreich wären, um nur eine Hand voll Peers untereinander zu verbinden. Desweiteren wäre die Auflösung auf AS-Ebene viel zu grob, da es schon wichtig wäre, mehrere Peers innerhalb eines Autonomen Systems möglichst günstig zu vermitteln. Das Netz der Deutschen Telekom besteht zwar z.B. aus verschiedenen Subnetzen, allerdings bildet die Deutsche Telekom nur ein Autonomes System (siehe dazu Abbildung 6). Außerdem hat man herausgefunden, daß die Informationen, die man aus BGP-Routing-Tabellen erhält, oft sehr stark denen widersprechen, die man durch Traceroute-Aufrufe übermittelt bekommt, weil einzelne Router dann doch andere Vorstellungen von dem Routing entwickeln, bzw. widersprüchliche manuelle Anweisungen bekommen [HBc03].

Also muß zwangsläufig überprüft werden, welche Wege von den Paketen durch das Internet tatsächlich genommen werden und dazu gibt es Traceroute.

## 3.2 Traceroute

Traceroute ist ein Programm, welches schon seit den frühesten Anfängen von Unix existiert. Es nutzt das ICMP-Protokoll (Internet Control Message Protocol [8]), um den Pfad über die Router, den die IP-Pakete gehen, nachverfolgen zu können.

Nach dem Aufruf (*traceroute [IP-Adresse oder Hostnamen]*) sendet die Quelle immer wieder IP-Pakete an den Zielhost und erhöht jeweils den Time-To-Live-Zähler um eins. Zurück kommen für jeden Router, der zwischen Quelle und Zielhost liegt Fehlerpakete (sogenannte „ICMP

unreachables“), falls die TTL bei ihm abgelaufen ist. In der Rückmeldung ist natürlich die IP-Adresse des Routers enthalten. Pro TTL wird meistens drei mal die Zeit zwischen dem absenden und erhalten des Paketes der Antwort gemessen.

Die TTL-Zeit kann man auch als Radius verstehen.

Das Programm Traceroute existiert in irgendeiner Form auf allen gängigen Betriebssystemen. Unter Windows findet man es unter dem Namen *tracert* und ab Windows XP gibt es zusätzlich noch ein Programm namens *pathping*, welches ein paar Erweiterungen des alten Traceroutes zulässt.

Von dem Traceroute-Programm unter Unix existieren viele verschiedene Implementationen, welche in der Regel immer unter dem Namen *traceroute* zu finden sind. Trotz der minimalen Unterschiedlichkeiten zwischen den Versionen, halten diese immer einen kleinsten gemeinsamen Nenner ein. So kennen alle Implementationen den Parameter „-n“ (bei Windows „-d“), welcher bewirkt, daß nur die IP-Adressen ausgegeben werden und nicht die Hostnamen. Dies ist sehr wichtig, wenn die Ausgabe von Traceroute verarbeitet werden soll und da sind die Hostnamen eher hinderlich. Außerdem müßte sonst für jeden Hostnamen ein Reverse-DNS-Lookup gemacht werden, der Zeit kostet. Man könnte sogar den Aufruf von Traceroute durch einige andere Parameter noch weiter verkürzen, z.B. wäre es für manche Zwecke nicht notwendig, alle drei Zeitmessungen zu machen - eine würde reichen. Jedoch muß man auf den kleinsten gemeinsamen Nenner achten.

Hier ein Beispiel eines Traceroute-Aufrufs:

```
#~: traceroute qui-gon.upb.de
traceroute to qui-gon.upb.de (131.234.36.71), 30 hops max, 38 byte packets
 1 217.0.116.106 (217.0.116.106) 66.522 ms 28.597 ms 20.529 ms
 2 217.0.71.86 (217.0.71.86) 19.608 ms 12.432 ms 86.579 ms
 3 h-eb1.H.DE.net.DTAG.DE (62.154.49.170) 55.974 ms 206.636 ms 79.293 ms
 4 ar-hannover3-p01-0.g-win.dfn.de (188.1.62.1) 103.062 ms 27.626 ms 13.046 ms
 5 cr-hannover1-ge5-1.g-win.dfn.de (188.1.88.1) 13.218 ms 13.609 ms 13.491 ms
 6 cr-essen1-po0-0.g-win.dfn.de (188.1.18.49) 30.292 ms 25.431 ms 29.571 ms
 7 ar-bielefeld1-po5-0.g-win.dfn.de (188.1.86.78) 28.234 ms 45.908 ms 28.099 ms
 8 pbj-loop.uni-paderborn.de (131.234.1.129) 28.682 ms 32.084 ms 198.727 ms
 9 netz1-1.uni-paderborn.de (131.234.1.2) 126.471 ms 74.448 ms 76.097 ms
10 qui-gon.cs.upb.de (131.234.36.71) 57.715 ms 35.723 ms 39.677 ms
```

Abb. 3: Traceroute-Aufruf

Gelesen würde es als IP-Pfad  $\mathbf{P} = \langle 217.0.116.106, 217.0.71.86, \dots, 131.234.36.71 \rangle$

### 3.2.1 Problematiken von Traceroute

Problematisch wird es, wenn der Zielhost mit einer Firewall geschützt ist und dieser keine ICMP-Pakete zurücksendet, bzw. sogar ein Router auf dem Weg dorthin keine ICMP-Nachrichten verarbeiten möchte und sie einfach nicht beantwortet. Da Router jedoch auf das ICMP-Protokoll angewiesen sind, ist es sehr unwahrscheinlich, daß dies passiert. Allerdings kann es passieren, daß wenn der Router stark ausgelastet ist, er einfach nicht dazu kommt innerhalb der maximalen Zeit das ICMP-Paket zu beantworten, weil diese auch eine sehr niedrige Priorität haben.

Der normale Fall ist es jedoch, daß wenn die ICMP-Pakete die maximale Zeit überschreiten, der Router mit der  $TTL - 1$  der letzte Router war und der Endknoten nicht ansprechbar ist, oder die Firewall des Endknoten ICMP-Pakete verbietet. In diesem Fall werden „Sterne“ von Traceroute angezeigt.

Allerdings kann es auch in seltenen Fällen vorkommen, daß es mitten im Traceroute zu Timeouts kommt, wie in Abb. 4 zu sehen ist. Diese müssen dann einfach übergangen werden.

```
#~: traceroute -n 217.20.118.113
traceroute to 217.20.118.113 (217.20.118.113), 30 hops max, 40 byte packets
 1 10.0.1.1 1.575 ms 0.906 ms 0.877 ms
 2 * * *
 3 145.253.4.149 20.303 ms 14.509 ms 13.215 ms
 4 145.254.12.241 13.794 ms 13.265 ms 13.115 ms
 5 145.254.16.174 14.763 ms 15.108 ms 15.249 ms
 6 80.81.193.22 16.804 ms 17.222 ms 17.173 ms
 7 134.222.227.137 20.839 ms 21.922 ms 27.288 ms
 8 134.222.104.11 25.933 ms 21.121 ms 15.211 ms
 9 134.222.104.58 16.842 ms 19.188 ms 15.134 ms
10 * * *
11 * * *
12 * * *
```

Abb. 4: Problemhaftes Traceroute

### 3.2.2 Bereinigen von Traceroutes

Die Lösung um möglichst viele Fehlerfälle abzufangen, die von einem Traceroute-Aufruf erzeugt werden können, beinhaltet folgende Punkte:

- Falls die letzte IP-Adresse ungleich der Ziel-IP-Adresse, dann ergänze den Pfad um die Ziel-IP-Adresse.
- Falls zwischendurch Sterne auftauchen, ignoriere sie. Füge sie nicht zum Pfad hinzu!
- Falls dreimal hintereinander drei Sterne pro TTL gemeldet werden, beende Traceroute.
- Falls nach den Korrekturen nur noch eine IP-Adresse übrig ist und zwar die, die hinzugefügt wurde, dann verwirfe den Traceroute. Schlussfolgerung: Der Rechner, welcher

dieses Traceroute ausführt, ist von einer Firewall umgeben und darf keine ICMP-Pakete nach außen verschicken.

Anschließend müssen noch interne IP-Adressen entfernt werden. Für Intranets sind bestimmte IP-Bereiche reserviert [9]:

Von	Bis
10.0.0.0	10.255.255.255
172.16.0.0	172.31.255.255
192.168.0.0	192.168.255.255

Diese können bei Masquerading Firewalls [10] im Traceroute vorkommen und müssen herausgefiltert werden, da sie falsche Rückschlüsse zulassen würden, z.B. wenn zwei LAN-Netzwerke das Netzwerk *192.168.1.0-255* benutzen und diese im Traceroute enthalten sind, könnte man irrtümlicherweise annehmen, die beiden wären untereinander verbunden.

Um den Pfad komplett zu machen, muß nun noch die eigene IP-Adresse vorne angesetzt werden. Allerdings entsteht dadurch ein weiteres Problem durch Masquerading Firewalls. Der Rechner ist sich seiner eigenen IP-Adresse nach Aussen nicht bewusst. Der Rechner kann die IP-Adresse *10.0.1.1* haben (siehe Abb. 4), ist aber im Internet unter einer anderen IP-Adresse (z.B. *145.253.4.149*) sichtbar. Dieses Problem kann man dadurch lösen, daß man einen anderen Rechner im Internet fragt, der garantiert nicht im eigenen LAN ist, mit welcher IP-Adresse man sich bei ihm gemeldet hat. Deswegen darf ein Rechner nie seine eigene IP-Adresse bestimmen.

Nach diesen Verbesserungen würde der „bereinigte“ Pfad aus Abbildung 4) nun folgendermaßen lauten, wenn der Aufrufer die IP-Adresse *84.139.75.4* hätte:

$P = \langle 84.139.75.4, 145.253.4.149, 145.254.12.241 \dots 217.20.118.113 \rangle$
---

### 3.2.3 Varianzen in Roundtrip-Zeiten

Viele Ansätze zur Bestimmung von geeigneten Verbindungen basieren auf den Messungen der Zeit zwischen dem Abschicken und Empfangen der ICMP-Pakete, was auch mit dem Befehl *Ping* möglich ist. Allerdings haben meine Experimente damit gezeigt, daß diese Zeiträume von zu vielen Faktoren abhängen:

- Auslastung der eigenen Internet-Anbindung, bzw. auch der Verkehr auf dem eigenem LAN
- Rechenzeitauslastung auf dem eigenen Rechner
- Momentane Tageszeit

- Momentane Auslastung der Router zwischen Quelle und Ziel (bei zu hoher Auslastung, werden ICMP-Pakete niedriger priorisiert)
- Granularität des Zeitgebers der Hardware

Man kann u.a. beobachten, daß Pakete zu dem Router mit TTL  $n+1$  manchmal weniger lange benötigen, als zu dem Router mit TTL  $n$ . Die Zeiten von den drei Messungen können genauso stark voneinander abweichen. Selbst das Mitteln der Werte würde kaum mehr verlässliche Informationen bringen.

Deshalb ist die Laufzeit der Pakete nicht geeignet, um auch nur irgendeine Distanzaussage zu tätigen und nur die reinen Verbindungsinformationen, die aus dem Traceroute resultieren, sind für die Bestimmung der Netzwerktopologie verwertbar.

### 3.2.4 Varianzen in Routen

Jedoch nicht nur die Zeiten, sondern auch die IP-Adressen in den Routen, welche zur identischen IP-Adresse führen sollen, variieren stark. Dies kommt daher, daß alle größeren Provider Mechanismen nutzen, um auf Netzauslastung reagieren zu können und betreiben eine Lastverteilung auf mehrere Router, die natürlich auch verschiedene IP-Adressen haben.

Für die Peers ist es jedoch egal, ob die Pakete über Router 1 oder Router 2 des Providers gehen, zumal diese sich im gleichen Netzwerk, meist sogar nebeneinander, im gleichen Raum befinden. Dies sind also redundante Informationen, die man zwar nicht benötigt, welche aber Adjazenz-Matrizen von Verbindungsgraphen ganz schön „aufblasen“ können.

Dies macht es notwendig solche Situationen zu erkennen, um nicht alle Varianten solcher Routen abbilden zu müssen. Was sich anbieten würde, wären dabei z.B. die Autonomen Systeme und deren Nummern. Jedoch wäre diese Zusammenfassung viel zu ungenau. So würden beispielsweise sämtliche Universitäten in Deutschland die über das Deutschen Forschungsnetz (DFN) angeschlossen sind, nur noch als ein Knoten erscheinen. Einen Kompromiss kann man durch Anfragen bei den Regionalen IP Registraren (RIRs) erreichen.

## 3.3 Gruppieren von IP-Adressen

In den frühen Anfängen des Internets waren die IP-Adressen an sich schon in Bereiche eingeteilt. Man unterschied zwischen den verschiedenen Netzgrößen A-E und je nach Größe wurden diese auf die wenigen Teilnehmer des damaligen Internets verteilt. Auch heute hat z.B. die Firma Ford noch ein komplettes Klasse-A-Netz (19.x.x.x) inne, welches 16646144 IP-Adressen umfasst, die diese Firma während ihrer gesamten Existenz wohl nie benötigen wird. Klasse-A-

Netze findet man im Bereich zwischen *0.0.0.0* und *127.255.255.255*.

Desweiteren gab es noch Klasse-B-Netze, wie das der Uni-Paderborn, welches über 65024 IP-Adressen verfügt (*128.0.0.0 - 191.255.255.255*) und Klasse-C-Netze mit 254 IP-Adressen (*192.0.0.0 - 223.255.255.255*). Alle darüber hinaus (D und E) wurden für spezielle Verwendungszwecke wie Multicast reserviert. Auch das Routing richtete sich hierarchisch nach diesen Bereichen. Durch Netzmasken wurden die Netze in weitere, kleinere Subnetze eingeteilt.

Als das Internet immer mehr an Größe gewann, wurde schnell klar, daß man nun sparsamer mit der Vergabe von IP-Bereichen umgehen muß und die drei Klassengesellschaft der IP-Netze wurde durch das CIDR (Classless Inter-Domain Routing) abgelöst [11, 12].

Heutzutage hat ein Netzbereich eine Grundadresse und eine einschränkende Bitmaske, die als dezimaler Wert hinten angefügt wird, z.B. beschreibt das Netz der Universität Paderborn die CIDR *131.234.0.0/16*.

Da nun die einzelnen Netzbereiche beliebig einteilbar sind, muß dieses Chaos auch irgendwie verwaltet werden.

### 3.3.1 IP-Registriere

Um die Vergabe von IP-Adressen kümmert sich der Dachverband IANA (Internet Assigned Numbers Authority) und dieser verteilt Adressbereiche an die regionale Ebene, die RIRs (Regionale Internet-Registries), welche seit 2003 in der NRO (Number Resource Organization) organisiert sind. Es gehören dazu:

- RIPE - Réseaux IP Européens (Europa)
- ARIN - American Registry For Internet Numbers (Nordamerika und Kanada)
- APNIC - Asia Pacific Network Information Centre (Asien und die Pazifik-Region)
- LACNIC - Latin American and Caribbean Internet Addresses Registry (Lateinamerika und die Karibik)
- AfriNIC - African Internet Numbers Registry IP Addresses (Afrika)

Außerdem gibt es noch kleinere Untervermittler wie z.B. der JPNIC (Japan), BRNIC (Brasilien), KRNIC (Korea), V6NIC (Für IPv6), die allerdings meist nur ein paar kleinere Netze verwalten.

Wenn ein IP-Registrant Adressblöcke an Kunden vergeben hat, kann man bei den Whois-Servern des jeweiligen RIRs über das Whois-Protokoll mehr Informationen über diesen vergebenen Bereich erfahren, z.B. den Namen des Besitzers oder die Größe des vergebenen Bereichs.

### 3.3.2 Whois-Protokoll

Das Whois-Protokoll ist denkbar einfach [13]. Unter Unix kann man einfach den Befehl *whois* benutzen, um eine Abfrage zu starten. Whois-Server sind meistens mit dem prefix „whois“ plus dem zuständigen RIR erreichbar (z.B. *whois.ripe.net*). Wenn man also herausbekommen will, zu welchem Block eine gewisse IP-Adresse gehört (hier eine IP-Adresse aus dem Netzwerk der Uni-Paderborn), dann führt man den Befehl aus Abbildung 5 aus. Das komplette Ergebnis kann

```
#~: whois -h whois.ripe.net 131.234.22.30
[... ]
inetnum:      131.234.0.0 - 131.234.255.255
netname:      UNIPADERBORN
[... ]
```

Abb. 5: Ausschnitt aus der RIPE-Information für das Netz der Universität Paderborn

im Anhang A betrachtet werden.

Dank dieser Auskunft weiß man nun, welche IP-Adressen zu der Uni-Paderborn gehören. Allerdings weiß man jedoch nicht, ob vielleicht nicht noch andere Einträge existieren, die auch zur Uni-Paderborn gehören, d.h. ob die Uni-Paderborn über mehrere IP-Bereich verfügt. Dies ist z.B. bei der Deutschen Telekom der Fall, welche verschiedene Netzeinträge bei RIPE unterhält. Nun hat allerdings jeder RIR eine eigene Implementation der Datenbank und auch die Felder, die zurückgeliefert werden, unterscheiden sich. Bei Arin heißen die relevanten Felder z.B. NetRange und NetName.

### 3.3.3 Rekursiver Whois

Da nur eine halbwegs aktuelle Liste existiert [IAN05], in der aufgeführt wird, welche IP-Adressen zu welchem RIR gehören und die Informationen auf den Whois-Servern meist sehr viel aktueller als diese Liste sind, empfiehlt sich eine rekursive Abfrage der Whois-Daten.

Am bewährtesten ist es, zuerst bei ARIN eine Anfrage zu starten. Dort wird, falls ein anderer RIR (z.B. APNIC oder RIPE) für diese IP zuständig ist, auf diesen verwiesen. Beim APNIC kann es jedoch wieder passieren, daß man z.B. auf den JPNIC verwiesen wird. Da der APNIC die Daten des JPNIC spiegelt, kann man sich die Anfrage beim JPNIC sparen, allerdings muß man das auch wissen. Meistens muß also nicht tiefer als bis zu einer Rekursionstiefe 2 geforscht werden, um den richtigen RIR zu finden, außer bei einem Sonderfall: ARIN. Eine Besonderheit des ARIN-Servers ist, daß es verschachtelte Zuordnungen von IP-Bereichen gibt, die aufgelöst werden müssen. Tiefer wird auf dieses Prozedere in Kapitel 6.1.3 eingegangen.

### 3.3.4 Cachen der Whois-Anfragen

Es bietet sich an, um die Datenbanken der Registrare nicht zu überfordern, bzw. auch um eigenen Traffic einzusparen, die Anfragen an die einzelnen Whois-Server zu cachen. Das einzige Problem dabei ist nur: Wieviel Unterregistrator kann ein IP-Bereich haben? Ein IP-Bereich kann auch mehrfach untervermietet sein und man will so genau wie möglich herausfinden, in welchem Netzwerk sich der jeweilige Peer befindet.

Bei PeerNear wurde das Problem folgendermaßen gelöst: Wenn ein Netz schon einmal unterteilt war (unterteilt war), bzw. eine Untermenge eines anderen IP-Bereichs ist, dann führt man sicherheitshalber noch einmal eine spezielle Abfrage durch und nimmt den kleinsten IP-Bereich der dabei zurückgeliefert wurde als das Netz, worin sich diese IP-Adresse befindet.

## 3.4 Erweitertes Traceroute

Wenn man dies alles berücksichtigt, kommt man auf ein erweitertes Traceroute, welches nicht nur die einzelnen IP-Adressen pro TTL auflistet, sondern auch die zugehörigen Netznamen.

Als Vorstufe zu PeerNear habe ich ein Tool geschrieben, welches einen solchen erweiterten Traceroute durchführt. Eine Beispielausgabe ist in Abbildung 6 zu sehen.

IP	ASNAME	Netname	IP-Range
131.234.36.2	[ 'AS680', 'AS1275' ]	[ 'UNIPADERBORN' ]	[ '131.234.0.0 - 131.234.255.255' ]
131.234.1.1	[ 'AS680', 'AS1275' ]	[ 'UNIPADERBORN' ]	[ '131.234.0.0 - 131.234.255.255' ]
188.1.44.193	[ 'AS680' ]	[ 'WIN-IP' ]	[ '188.1.0.0 - 188.1.255.255' ]
188.1.86.77	[ 'AS680' ]	[ 'WIN-IP' ]	[ '188.1.0.0 - 188.1.255.255' ]
188.1.18.233	[ 'AS680' ]	[ 'WIN-IP' ]	[ '188.1.0.0 - 188.1.255.255' ]
188.1.18.206	[ 'AS680' ]	[ 'WIN-IP' ]	[ '188.1.0.0 - 188.1.255.255' ]
188.1.74.2	[ 'AS680' ]	[ 'WIN-IP' ]	[ '188.1.0.0 - 188.1.255.255' ]
62.156.138.185	[ 'AS3320' ]	[ 'DTAG-BB11' ]	[ '62.156.132.0 - 62.156.140.255' ]
62.154.107.70	[ 'AS3320' ]	[ 'DTAG-BB16' ]	[ '62.154.0.0 - 62.154.127.255' ]
217.237.154.213	[ 'AS3320' ]	[ 'DTAG-DIAL15' ]	[ '217.224.0.0 - 217.237.161.47' ]

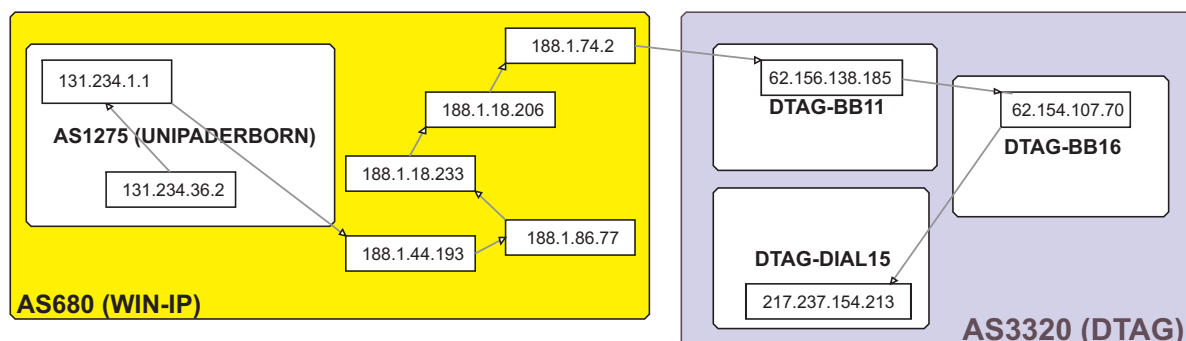


Abb. 6: Ausgabe des erweiterten Traceroute mit Visualisierung

## 3.5 Internet-Grapheigenschaften

Um eine Idee von den Grapheigenschaften von Internet-Graphen zu bekommen, habe ich zuvor einige empirische Untersuchungen an den Ergebnissen des Skitterprojekts [HPMc02] vorgenommen.

Skitter ist ein Projekt, bei dem verschiedene Rechner, die auf der ganzen Welt verteilt sind, die ganze Zeit nur Traceroutes zu zufälligen IP-Adressen aufrufen, um dadurch den kompletten Internet-Graph kartographieren zu können. Diese Daten werden zu einer Adjazenzmatrix zusammengeführt, die den bisher erforschten Internet-Graph widerspiegelt. Diese Adjazenzmatrix ist auf der Homepage in zwei Varianten erhältlich:

**AS-Graph** Dort wurden die IP-Adressen aus den zuvor erhaltenen Traceroutes den einzelnen Autonomen Systemen zugeordnet, bevor sie in den Graph eingefügt wurden. Ein Knoten im Graph ist also gleich einem Autonomen System.

**Routing-Graph** Hier wurden die Router-IP-Adressen belassen, jedoch wurden diese noch zusätzlich bereinigt. Z.B. wurden Router mit mehreren IP-Adressen, zu einem Knoten vereinigt und alle Knoten mit Gesamtgrad=0 gelöscht.

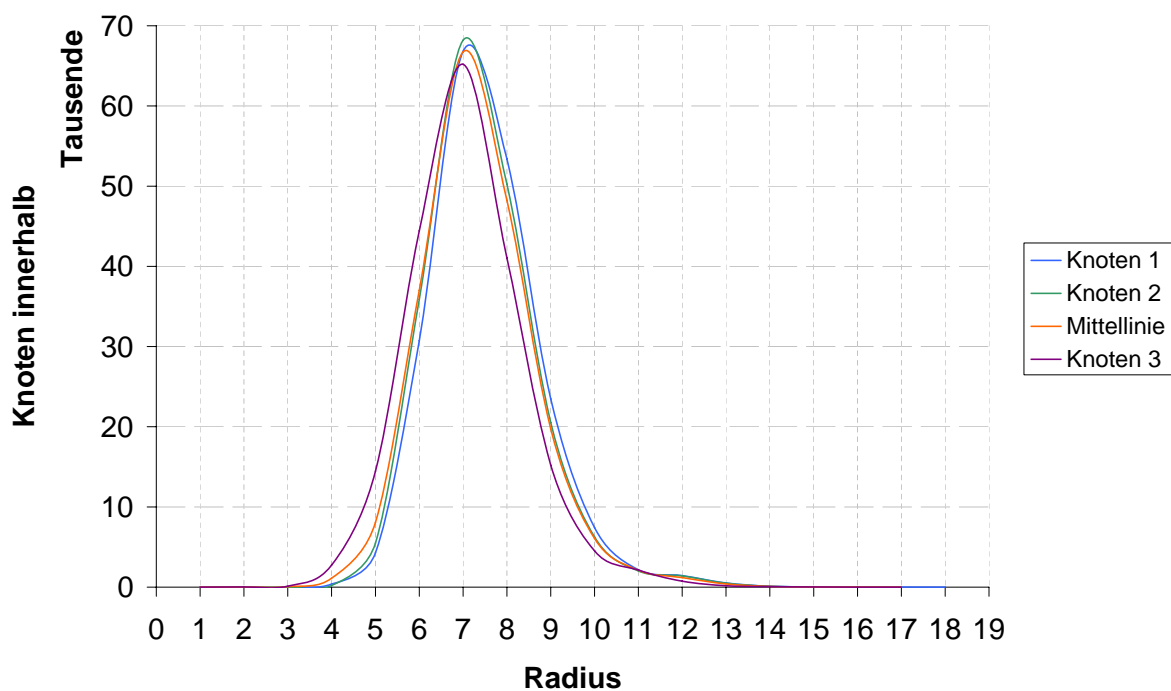
Da bei dem AS-Graphen schon viele Informationen verloren gegangen sind, habe ich mich bei meinen Untersuchungen für den Routing-Graph entschieden, welcher unter [http://www.caida.org/tools/measurement/skitter/router\\_topology/](http://www.caida.org/tools/measurement/skitter/router_topology/) zu finden ist.

Von dieser Router-Adjazenzmatrix gibt es eine gerichtete und eine ungerichtete Variante, wobei ich mich für erstere entschieden habe, da Traceroutes immer gerichtet sind und keine relevanten Informationen verloren gehen können.

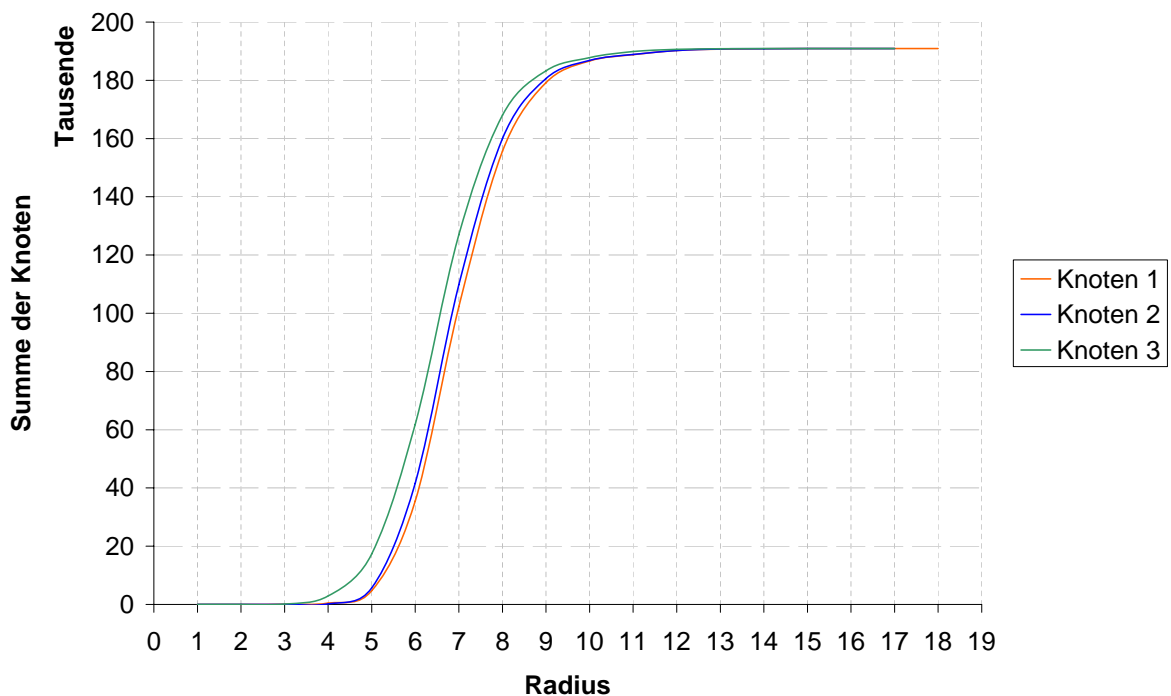
Insgesamt befanden sich in diesem Graph 192244 Knoten und diese Knoten hatten einen durchschnittlichen Gesamtgrad von 6,62

### 3.5.1 Nachbarschaftszuwachs

Als erstes habe ich die Zuwachseigenschaften des Graphen von drei zufälligen Knoten aus gemessen. Dazu habe ich die Skitter-Adjazenzmatrix eingelesen, von jedem der drei Knoten eine Breitensuche [CLR97] gestartet und protokolliert, wieviele Knoten sich im Radius  $r$  um diese Knoten befinden. Dazu habe ich von dem Breitensuchenergebnis nur die Knoten genommen, die in der Distanz  $r$  gefunden wurde. Beim zweiten Graph habe ich hingegen auch alle bisherigen gefundenen Knoten bis zum Radius  $r$  hinzugenommen und dargestellt. Den Parameter  $r$  habe ich dabei von 1 an, solange laufen lassen, bis alle Knoten im Graph erreicht wurden. Die



(a) Diskrete Nachbarschaften



(b) Summe der Nachbarschaft

Abb. 7: Ergebnisse der Analyse des Wachstums nach Radius

Ergebnisse sind in Abbildung 7 zu sehen.

Wie man leicht sehen kann, weicht der Wachstumsverlauf aller drei Knoten kaum voneinander ab. Es ist ein exponentielles Wachstum zu sehen, wobei das Maximum bei einem Umkreisradius von 7 liegt und danach wieder exponentiell abnimmt. Bei einem Umkreisradius von 18 kommen keinerlei Knoten mehr hinzu. Das heißt wir können davon ausgehen, daß dieser Graph einen maximalen Durchmesser von 18 hat.

### 3.5.2 Pareto-Verteilung

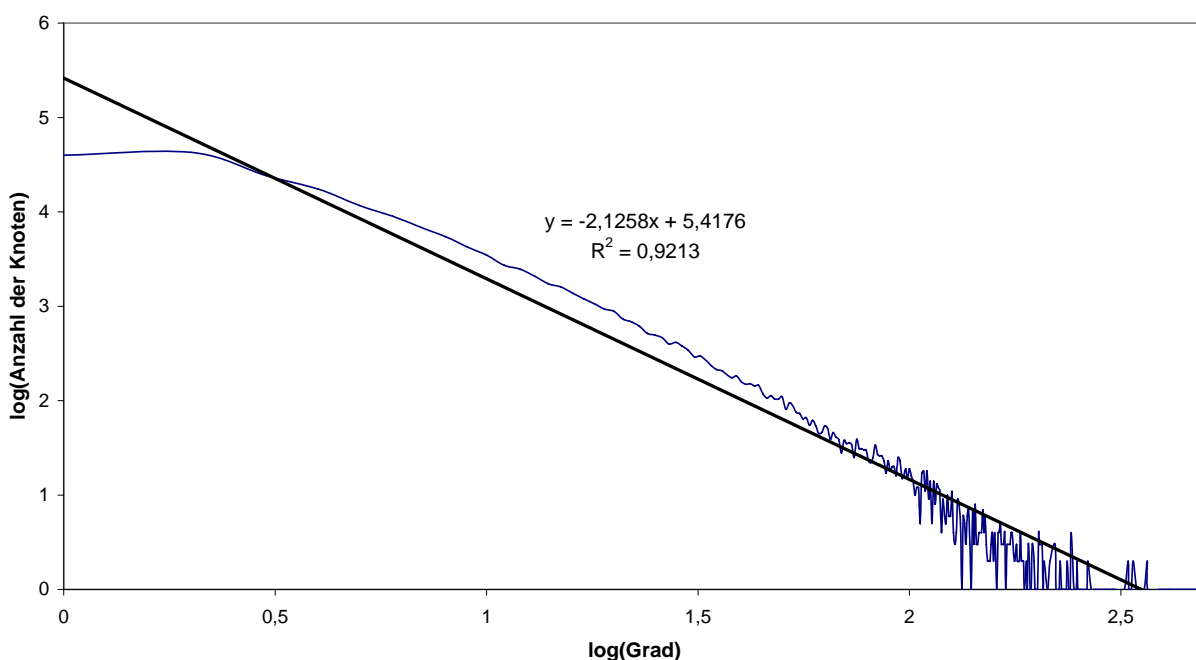


Abb. 8: Verteilung der Knotengrade

Desweiteren habe ich die Verteilung der Knotengrade untersucht (siehe Abb. 8). Bei der Abbildung auf einer doppeltlogarithmischen Skala fällt auf, daß es sich bei der Verteilung der Knotengrade um eine Pareto-Verteilung und somit um einen Power-Law-Graphen handelt.

Daß das Internet ein Power-Law-Graph ist, wurde als erstes von Faloutsos [FFF99] vermutet und nach einigen Diskussionen erneut von Jaiswal [JRT04] bestätigt.

Das Power-Law ist wie folgt definiert ( $c$  und  $i$  seien konstant):

$$y(x) = \frac{c}{x^i} \quad \Rightarrow \quad \ln(y) = \ln\left(\frac{c}{x^i}\right) = -i \cdot \ln(x) + \ln(c)$$

Daraus folgt, wenn man eine doppelt logarithmische Skalierung des Graphen vornimmt, eine Linie hindurch legt und dann von dieser die Steigung bestimmt, erhält man den Parameter  $-i$

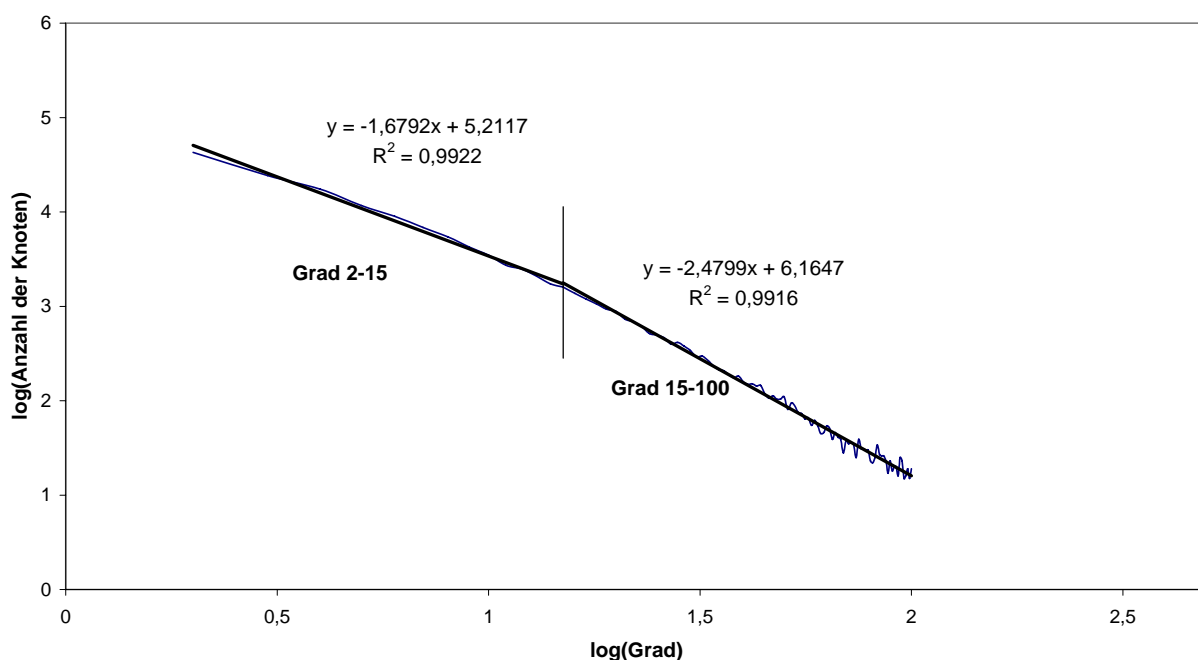


Abb. 9: Überlappen der Pareto-Verteilungen

dieses Graphen. Die wahrscheinliche Häufung von Knotengrad  $x$  ist dann proportional zu  $\frac{1}{x^i}$ .

Wie man in der Abbildung 8 auch sehen kann, geht die Linie nicht genau durch den Graph hindurch und weist „nur“ ein Bestimmtheitsmaß von  $R^2 = 0,9213$  auf. Die Steigung beträgt hier gemäß der Geradengleichung übrigens  $i = -2,1258$ .

Wenn man den Graph allerdings in die Grade 2-15 und 15-100 einteilt und dann nochmal versucht eine Linie durch die beiden Abschnitte zu legen, sieht man in Abbildung 9, daß diese perfekt passen. Schlußzufolgern ist, daß sich mehrere Pareto-Verteilungen in diesem Graphen überlagern. Eine hat  $i = -1,6792$  für die Grade 2-15 und die andere  $i = -2,4799$  für die Grade 15-100.

### 3.5.3 Small-World-Eigenschaft

In Kapitel 2.6 wurde die Frage aufgeworfen, ob der Internet-Routing-Graph die Small-World-Anforderung (kleiner Durchmesser und geringe Dichte) erfüllt. Wir haben nun einen Durchmesser von 18 bei ca. 200000 Knoten feststellen können, welcher somit ungefähr bei  $\log_2(n)$  bei  $n$  Knoten liegt. Außerdem sieht man an der überlappenden Pareto-Verteilung, daß der Großteil der Knoten einen Kantengrad kleiner 15 hat, welches ebenfalls ein Indiz für Small-World-Graphen ist. Zu ähnlichen Ergebnissen ist übrigens auch Jin [JB02] gekommen, welcher den Internet-Routing-Graph auf die Small-World-Eigenschaften untersucht hatte. Deshalb kann die Frage, ob das Internet dem Small-World-Phänomen unterliegt, eindeutig positiv beantwortet werden.

## 4 Planen von Traceroute-Aufrufen

Um die Distanzen der Peers zueinander ermitteln zu können, gibt es verschiedene Möglichkeiten: man könnte zum Beispiel jeden Peer veranlassen einen Traceroute zu jedem anderem Peer durchzuführen. Da aber ein Traceroute-Aufruf bis zu mehrere Minuten dauern kann und die Anzahl der Traceroute-Aufrufe in  $n(n - 1) \in O(n^2)$  bei  $n$  Knoten liegen würde, ist davon abzuraten. Deshalb musste ein Approximationsalgorithmus entwickelt werden, der die Anzahl der Traceroute-Aufrufe reduzieren kann. Dazu müsste aber das Problem spezifiziert, bzw. erst einmal modelliert werden.

Folgende Annahmen kann man aus dem vorausgegangenen Kapitel ableiten, um die Basis für einen Algorithmus zu schaffen:

- Traceroutes sind zwar nicht IP-, aber annähernd Netzsymmetrisch.
- Peers können das Routing nicht beeinflussen, sie können lediglich via Traceroute die Routen erfragen, somit müssen die Routen zwischen Peers als optimal hingenommen werden.
- Innerhalb eines Autonomen Systems gehen die Kosten für das Routings gegen 0.
- Verbindungen innerhalb des Autonomen Systems sind schneller, als die zu externen Netzen
- Falls Verbindungen zu externen Netzen notwendig werden, versuche die Anzahl der Router und Verbindungen zu minimieren

Daraus folgt: man muß versuchen den Graphen zwischen den Routernetzen mit Traceroutes zu lernen, wobei die Traceroutes nicht IP-bezogen auszuführen sind, sondern netzbezogen.

### 4.1 Erlernen des Graphen

Da die Performanz auf den einzelnen Leitungen nicht interessiert und sie keine Aussagekraft hat, versieht man jede einzelne Kante mit dem Gewicht 1. Ein Knoten des Graphen ist dann jeweils ein Netz, welches einen gewissen IP-Bereich umfaßt. Diese Zuordnung erhält man über die Whois-Informationen.

Man muß nun den Graph erlernen und dies mit möglichst wenig Traceroute-Aufrufen. So entsteht ein interaktives Spiel:

Wiederhole solange, bis keine Verbesserungen am Graphen mehr sinnvoll sind:

- Mache Traceroute zu dem Knoten von dem vermutet wird, daß dieser Traceroute am vielversprechendsten ist
- Bilde diesen Traceroute in der Adjazenzmatrix des Verbindungsgraphen ab
- Berechne, welche Traceroutes als nächstes durchgeführt werden sollen

Nun muß ein Algorithmus gefunden werden, der die Traceroute-Aufrufe koordinieren kann, bzw. bestimmt, welche am sinnvollsten sind.

## 4.2 Einführung des Begriffs Zelt als Erweiterung des Graphbegriffs

Zur Modellierung eines solchen Algorithmus wurde eigens für diesen Zweck ein neuer Ansatz namens „Zelttheorie“ entwickelt.

Da der Graph zwischen den Peers wie ein „Zelt“ aufgespannt werden soll, kann man ihn auch als ein solches verstehen. Dabei werden die Verbindungen zwischen den Knoten als Seile und Stangen dargestellt.

Nun stelle man sich vor, es stehe jemand mit dem Plan des Zeltes, so wie es einmal aussehen soll (hier: der Internet-Graph, so wie er ist), ein wenig entfernt und man selber versuche dieses Zelt mit so wenig Stangen und Seilen wie nötig, möglichst originalgetreu aufzubauen. Dabei sagt derjenige, der den Plan in der Hand hält, bei jeder Stange die eingefügt wird, wie lang diese Stange im Originalplan war und über welche Seile diese Stange mit anderen Stangen verbunden werden soll. Der Aufbauer weiß, daß er in einer Region aufhören kann weitere Stangen einzubauen, wenn zwischen den Stangen und Seilen in dieser Region kein Spiel mehr besteht und somit dieser Teil des Zeltes stabil ist. Dies wird solange gemacht, bis das komplette Zelt stabil ist. Natürlich kann er danach auch noch weitere Stangen einfügen, jedoch wäre dies nur Materialverschwendung.

**Definition** Ein Zelt  $T$  besteht aus drei Mengen: Knoten, Seilen und Stangen  $T = (V, R, S)$ . In einem gerichteten Zelt bestehen Stangen und Seile aus Knotenpaaren, in einem ungerichteten Zelt bestehen sie aus zwei-elementigen Knotenmengen.

In gewichteten Zelten erhalten die Seile  $r$  und die Stangen  $s$  ein Gewicht  $w(r)$ . In ungewichteten Zelten beträgt dieses Gewicht 1. Eine Stange mit zwei losen Enden zwischen  $u$  und  $v$  ist ein Seilpfad, welcher von  $u$  zu einem Knoten  $x$  führt, einem Stock der  $x$  und  $y$  verbindet und einem Seilpfad der von  $y$  nach  $v$  führt. Die minimale Distanz zwischen den beiden Enden ist die Länge der Stange, weniger der Länge der Seile.

Die maximale Distanz zwischen zwei Knoten wird bestimmt durch das kürzeste Seil zwischen

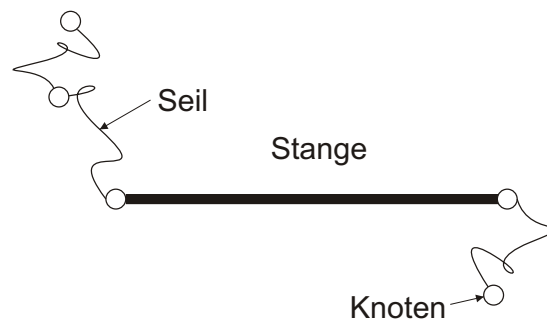


Abb. 10: Elemente der Zelttheorie

den beiden Knoten. Die minimale Distanz zwischen zwei Knoten ist die minimale Distanz einer Stange, die diese Knoten als lose Enden hat. Ein Zelt wird geschlossen genannt, wenn die Maximaldistanz immer länger als die Minimaldistanz zwischen zwei Knoten ist.

### 4.3 Traceroute im Sinne der Zelttheorie

Ein Traceroute ist ein Hybrid im Sinne der Zelttheorie. Er ist eine Stange, der einen Teil des Zeltes aufspannt, er besteht aber gleichzeitig auch aus Knoten und Seilen, wobei die Seile zusammen die gleiche Länge haben, wie die Stange selber. Man könnte es sich so vorstellen, daß die Seile an der Stange fixiert sind.



Abb. 11: Traceroute als fixiertes Seil an einer Stange

### 4.4 Traceroute Scheduler Algorithmus

Der Traceroute Scheduler Algorithmus stellt das Kernstück des PeerNear-Systems dar. Er ist dafür zuständig, eingehende Traceroutes zu verarbeiten, um anschliessend neue, vielsprechende Traceroutes den einzelnen Peers vorschlagen zu können.

Im wesentlichen benötigen wir dazu 4 Matrizen:

- All-Pair-Shortest-Pair-Matrix  $A$  (Zelttheorie: Seile, obere Schranke der Entfernung)
- Minimum-Distanz-Matrix  $B$  (Zelttheorie: Stangen, untere Schranke der Entfernung)
- Verbindungsmatrix  $C$  (Adjazenzmatrix)

- Differenzmatrix  $D = A - B$  (Zelttheorie: Spiel zwischen Seilen und Stangen, gibt den momentanen Lernfortschritt an)

#### 4.4.1 Einfügen eines Traceroutes in die A-Matrix

Die A-Matrix beinhaltet alle Distanzen von jedem Peer zum anderen. Sie stellt somit ein All-Pair-Shortest-Path-Problem dar. Eine Lösung dieses Problems bietet der Floyd-Warshall-Algorithmus [CLR97], welcher in  $O(n^3)$  liegt.

Dieser müsste vor jeder Berechnung der D-Matrix einmal durchgeführt werden. Wenn man dies nach jedem Einfügen eines Traceroutes machen möchte, läge man damit bei  $t$  Traceroutes in  $t \cdot O(n^3)$ .

Für eine Matrix A die anfänglich die Adjazenz-Matrix des Graphen enthält, sieht der Algorithmus folgendermassen aus:

```
for (k=0; k<n; k++)
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      A[i][j]=min(A[i][j], A[i][k]+A[k][j])
```

Folgende Probleme sprechen gegen den Floyd-Warshall-Algorithmus:

- Kubische Laufzeit
- Für jeden Traceroute der hinzugefügt wird, muß die Matrix neu aufgebaut werden

Jedoch kann man aufgrund der Natur des Verbindungsgraphen, bei dem alle Kanten unär und gerichtet sind, auch folgendes vorraussetzen:

Eine Kante  $(u, v)$  kann nur die Distanzen der Knoten verkürzen, die einen Weg eingehend zu  $u$  bzw. von  $v$  ausgehend besitzen.

Zu Realisieren ist dies durch zwei Aufrufe einer Breitensuche, einmal von  $u$  aus und einmal von  $v$  aus. Da wir das Minimum der Distanzen suchen, können wir die Tiefe der Breitensuche auf den maximalen Durchmesser des bisherigen Graphen begrenzen.

Eine weitere Optimierungsmöglichkeit ist, daß wir Kanten, die wir schon hinzugefügt haben, nicht noch einmal hinzufügen müssen.

Damit ein Traceroute in die A-Matrix eingefügt werden kann, muß er zuerst in seine Einzelkanten (Seile) zerteilt werden. Also z.B. bei einem Traceroute  $T = \langle 1, 2, 3, 4 \rangle \mapsto TT = \{(1, 2), (2, 3), (3, 4)\}$ .

Sei  $BFS(Startknoten, Richtung, maximaler\ Radius)$  eine Breitensuche, wobei  $Richtung \in \{0, 1\}$  angibt, ob die Suche in Kantenrichtung (=1) oder gegen sie (=0) verlaufen soll. Wenn ein maximaler Radius angegeben wird, breche die Suche bei dieser Breite ab. Rückgabe ist eine Liste von Tuplen mit den gefundenen Knoten und der Angabe in welchem Abstand zum Startknoten sie gefunden wurden.

Für jede Kante  $(u, v) \in TT$  führt man nun folgenden Algorithmus aus:

```
def addedge(u,v):
    if (C[u,v] == 1):          # Kante existiert schon -> exit
        return

    C[u,v]=1 # Füge Kante in Verbindungsgraphmatrix ein
    A[u,v]=1 # Füge Kante in A ein

    # BFS führt eine Breitensuche durch und gibt eine Liste von
    # Tuplen zurück: (Knoten, Distanz)
    # Durch maxradius kann die Tiefe der Breitensuche begrenzt werden

    # maxdist ist der aktuelle Maximale Durchmesser des Graphen

    for i, idist in BFS(u, forward=0, maxradius=maxdist):
        # für alle Knoten i im Umkreis maxdist von u aus rückwärts

        for j, jdist in BFS(v, forward=1, maxradius=maxdist):
            # für alle Knoten j im Umkreis maxdist von v aus vorwärts

            if i != j:          # falls nicht der gleiche Knoten
                dist=A[i,j]     # A[i,j]=minimum(A[i,j], idist+1+jdist)
                newdist=idist+1+jdist # idist ist die Entfernung i nach u
                if dist > newdist: # jdist ist die Entfernung j nach v
                    A[i,j]=newdist
                    dist=newdist

            if dist > maxdist: # wenn neue Entfernung größer als bisher
                maxdist=dist   # dann ist das der neue Maximaldurchmesser
```

Listing 1: Einfügen einer Kante in die A-Matrix

Dieser Algorithmus versucht sozusagen das Zelt an dem Seil, das gerade eingefügt wurde zusammenzuziehen. Natürlich kann er damit nur Seile beeinflussen, die auch mit dem neuen Seil verbunden sind. Jedoch gibt es nicht nur Seile im Zelt, sondern auch Stangen und man kann die Seile nicht weiter zusammenbringen, als es die Stangen erlauben. Deshalb muß man nun noch die Stangen „ausfahren“.

#### 4.4.2 Einfügen eines Traceroutes in die B-Matrix

Die B-Matrix macht genau das Gegenteil der A-Matrix. Sie versucht durch das einbringen von Stangen, das Zelt möglichst weit aufzuspannen.

Die Stange (der Traceroute) wird in die B-Matrix eingefügt, wobei die Länge des Traceroutes gleich der Länge der Stange ist. Dann werden alle Knoten, die über einen Pfad mit der eingefügten Stange verbunden sind, von temporären Hilfsseilen (siehe Abb. 12), voneinander weggedrückt. Es werden alle verbundenen Distanzen maximiert.

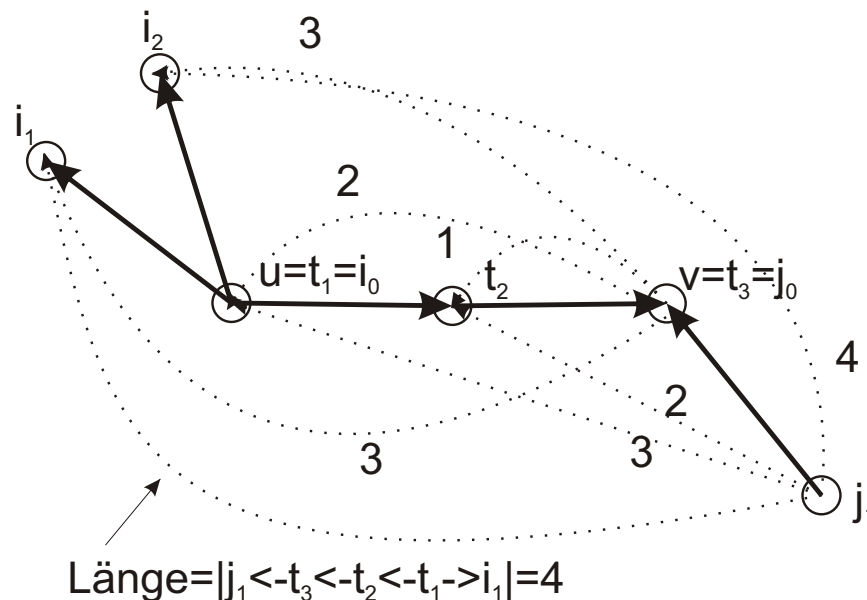


Abb. 12: Einfügen einer Stange in die B-Matrix

Folgender Algorithmus muß also für jeden Traceroute ausgeführt werden:

```
# Eingangsparameter thetrace ist eine Liste von Knoten
# hier schon transponiert in Indizes, z.B. [4,8,2,1]

def CalcB(thetrace):
    distuv=len(thetrace)-1 # Länge der Stange=Anzahl der Kanten im Traceroute
    u=thetrace[0] # Ausgangsknoten des Traceroute
    v=thetrace[-1] # Zielknoten des Traceroute

    B[u,v]=distuv # Trage Traceroute in die B Matrix ein

    d=len(thetrace) # Durchmesser der Breitensuchen

    for i,di in BFS(startnode=u, forward=1, maxradius=d):
        # Für alle Knoten i im Umkreis d von u aus vorwärts

        for j,dj in BFS(startnode=v, forward=0, maxradius=d):
            # Für alle Knoten j im Umkreis d von v aus rückwärts

            if i != j:
                # Falls nicht i und j nicht derselbe Knoten
                dist=B[i,j] # B[i,j]=maximum(B[i,j], distuv-di-dj)
                newdist=distuv-di-dj # wobei distuv=Länge des Stabs
                if dist < newdist: # und di die Länge des Seils von i nach u
                    B[i,j]=newdist # und dj die Länge des Seils von j nach v
```

Listing 2: Einfügen eines Traceroutes in die B-Matrix

Als Verbesserungen kann die Breitensuche auf die jeweilige Länge der Stange eingegrenzt werden, da die Stange keine Verbindungen, die weiter entfernt sind als der Stab selber lang ist, beeinflussen kann.

#### 4.4.3 Adjazenzmatrix C

Die Adjazenzmatrix C wird während des Einfügens der Kanten in die A-Matrix mitaufgebaut. Sie ist notwendig für später, falls ein Peer anfragen sollte, welche Peers sich in einem bestimmten Radius  $r$  um ihn befinden. Dann wird eine direkte Breitensuche auf diesem Verbindungsgraph ausgeführt mit dem Maximalradius  $mr = r$ . Außerdem liefert sie für die Breitensuchen in den A- und B-Algorithmen die Auskunft, welche Knoten mit den Stangen, bzw. mit den Seilen verbunden sind. Deswegen muß die Verbindungsmatrix auf jeden Fall mitgespeichert werden.

#### 4.4.4 Differenzmatrix D

Da die A-Matrix versucht das Zelt zusammenzuziehen und die B-Matrix, es aufzuspannen, ist logischerweise die Differenz zwischen den beiden, gleich dem Spielraum zwischen den Seilen und Stangen. Erst wenn die Differenz gleich 0 ist, besteht kein Spiel mehr, und das Zelt ist fertig aufgebaut.

Die Differenzmatrix  $D$  stellt also den momentanen Lernfortschritt dar, bzw. den momentanen Grad der Unwissenheit.

Es empfiehlt sich also alle Stangen einzufügen, die einen Eintrag in der D-Matrix größer 0 besitzen. Führe also nach jedem Einfügen einer Stange für alle  $D(u, v) > 0$  einen Traceroute  $Traceroute(u, v)$  aus.

Jedoch wäre es nicht besonders schlau, alle Traceroutes auf einmal zu starten, da sich aus den jeweiligen Ergebnissen wahrscheinlich die einen oder anderen Traceroutes erledigen würden.

Also veranlasst man nur einen Traceroute pro Peer, da die Peers parallel arbeiten können. Danach schicken die Peers ihre Ergebnisse zurück, die Ergebnisse werden in die A- und B-Matrix eingetragen, danach werden wieder Traceroutes zugewiesen usw. Solch einen Zyklus könnte man auch eine Spielrunde nennen.

Ein weiteres Problem ist, welche von den Matrixeinträge man für einen Knoten aussucht, um einen Traceroute dort hin zu machen, wenn sie in der B-Matrix den gleichen Wert haben? Welcher davon lohnt sich am meisten?

Der Zelttheorie nach lohnt sich die längste Stange, aber woher weiß man was die längste Stange ist, wenn man noch keinen Traceroute dorthin gemacht hat? Das kann man aus der A-Matrix ableiten. Der Eintrag der dort am größten ist, verspricht auch die längste Stange und somit auch

der größte Informationsgewinn zu werden.

Falls man nun noch mit einem gewissen Grad der Unwissenheit leben kann, kann man auch lediglich die Verbindungen überprüfen, die eine Schranke  $s$  nicht unterschreiten, also muß man nur noch die Traceroutes ausführen für die gilt  $D(u, v) \geq s$ , wobei  $s$  auch ein Parameter ist, womit man Traceroutes einsparen kann.

## 4.5 Erweiterung des Traceroute Scheduler Algorithmus

Ein großes Problem ist es, daß die Größe der A-Matrix quadratisch mit der Anzahl der Knoten ansteigt. Um dies zu begrenzen, wird ein weiterer Parameter eingeführt der den Radius und somit die Anzahl, wieviele Knoten zu einem Bereich gehören, begrenzt. Man teilt also den gesamten Graph in Bereiche ein, die alle als autonome Traceroute-Scheduler fungieren. Die autonomen Bereiche versieht man mit Indizes und fügt diese in einen bereichsübergreifenden Traceroute-Scheduler ein. Der Knoten, der als erstes in keinen existierenden Bereich fällt, eröffnet damit einen neuen Bereich und wird Repräsentant für diesen Bereich. Wenn ein Knoten in zwei oder mehrere Bereiche fallen würde, wird er dem Repräsentanten zugerechnet, zu dem er die kleinste Distanz hat.

Dabei ist darauf zu achten, daß bei dem Einfügen der Knoten des Traceroutes, mit dem letzten Knoten im Traceroute anzufangen ist, damit sicher gestellt wird, daß wenn ein neuer Bereich angelegt werden muß, dieser sich auch möglichst wenig mit existierenden Bereichen überschneidet.

Bevor Traceroutes eingefügt werden können, müssen diese nun zuerst in einen Traceroute umgewandelt werden, der den Weg durch den übergeordneten Graphen angibt. Danach muß der gleiche Ursprungs-Traceroute in Teil-Traceroutes unterteilt werden, die den Weg beschreiben, den dieser Traceroute in den jeweiligen untergeordneten Repräsentanten durchschreitet. Eine weitere hierarchische Unterteilung der untergeordneten Bereiche lohnt sich im übrigen nicht, da der Verwaltungsaufwand den Nutzen übersteigen würde.

Als Beispiel sollen zwei Traceroutes eingefügt werden, die schon von IP-Adressen auf Netzinizes übersetzt wurden:  $T_1 = \langle 1, 2, 3, 4 \rangle$  und danach  $T_2 = \langle 4, 5, 6, 7 \rangle$ . In Abbildung 13 und 14 sieht man die Ergebnisse und die zugehörigen Resultate wie sie in den Matrizen des Algorithmus abgebildet werden.

Wie man aus der  $D$ -Matrix des übergeordneten Traceroute-Schedulers erkennen kann, wäre die Traceroutes  $R_2 \rightarrow R_1$ ,  $R_3 \rightarrow R_1$  und  $R_3 \rightarrow R_2$  am vielversprechendsten, da sie noch keinerlei Informationen enthalten, ausser das dieser Weg unendlich (*inf*) erscheint.

$R_3$  hätte nun zwei Möglichkeiten, einen Traceroute zu starten ( $R_3 \rightarrow [R_1 \vee R_2]$ ). Er könnte

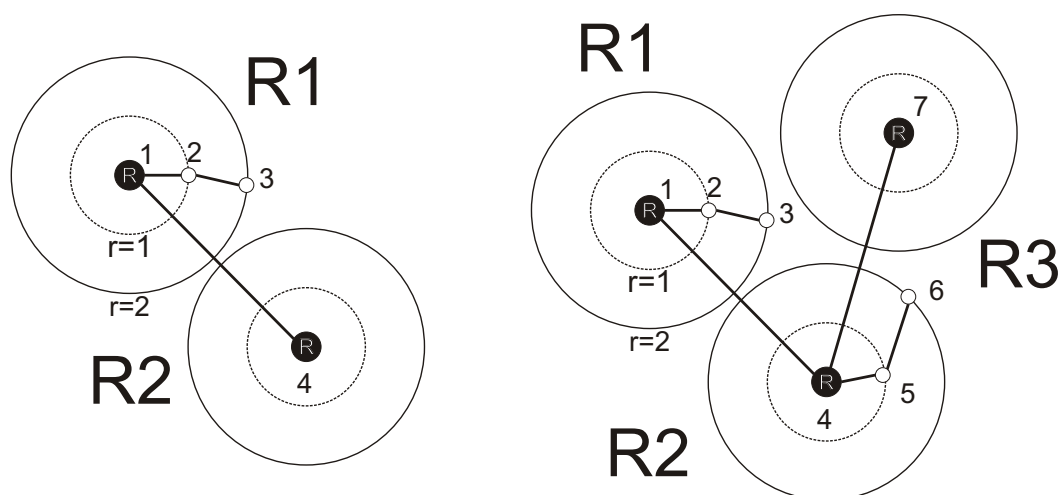


Abb. 13: Beispiel mit zwei Repräsentanten und Radius 2

entweder beide machen, oder aber sich für einen entscheiden. Da nun aber ein Aufruf, vielleicht die Situation schon so klären könnte, daß der zweite unnötig ist, wäre dies für die Laufzeit wesentlich besser. Dazu müsste sich  $R3$  entscheiden, welcher von beiden vielversprechender ist. Natürlich bringt der Traceroute am meisten Informationen, der schon in die andere Richtung (Hier  $R1 \rightarrow R3$ ) am längsten war, welches er aus der A-Matrix ablesen kann.

```

--Haupt-A-Matrix--
  1  2  3
1 inf inf inf
  2  1 inf inf
  3  2  1 inf
--Haupt-B-Matrix--
  1  2  3
1  0  0  0
  2  1  0  0
  3  0  1  0
----- C-Matrix -----
  1  2  3  4  5  6  7
1  0  1  0  0  0  0  0
2  1  0  1  0  0  0  0
3  0  1  0  1  0  0  0
4  0  0  1  0  1  0  0
5  0  0  0  1  0  1  0
6  0  0  0  0  1  0  1
7  0  0  0  0  0  1  0
--Haupt-D-Matrix--
  1  2  3
1 inf inf inf
  2  0 inf inf
  3  2  0 inf

Repräsentant 1
--A-Matrix R1--
  1  2  3
1 inf inf inf
  2  1 inf inf
  3  2  1 inf
--B-Matrix R1--
  1  2  3
1  0  0  0
  2  1  0  0
  3  2  1  0
--C-Matrix R1--
  1  2  3
1  0  1  0
  2  1  0  1
  3  0  1  0
--D-Matrix R1--
  1  2  3
1 inf inf inf
  2  0 inf inf
  3  0  0 inf

Repräsentant 4
--A-Matrix R2--
  4  5  6
4 inf inf inf
  5  1 inf inf
  6  2  1 inf
--B-Matrix R2--
  4  5  6
4  0  0  0
  5  1  0  0
  6  2  1  0
--C-Matrix R2--
  4  5  6
4  0  1  0
  5  1  0  1
  6  0  1  0
--D-Matrix R2--
  4  5  6
4 inf inf inf
  5  0 inf inf
  6  0  0 inf

```

Abb. 14: Zugehörige Matrizen

## 4.6 Performanz des Tracescheduler

Um die Effizienz des Algorithmus zu testen, wurde der Caida-Graph als Basis genommen, um eine Simulation durchzuführen. Dabei wurden nach und nach Pseudo-Peers in zufälligen Netzen hinzugefügt, und nach jedem eingefügten Netz wurde der Tracescheduler-Algorithmus angewendet. Der Repräsentanten-Radius wurde dabei von 2 bis 8 ( $r = 2 \dots 8$ ) variiert und die Traceroute-Aufrufe wurden durch eine Shortest-Path-Berechnung nach Dijkstra[CLR97] auf dem Caida-Graph [HPMc02] simuliert.

Das Programm, welches diese Simulationen durchgeführt hat, liegt im Sourcecode bei und kann jederzeit nachvollzogen werden. Das Programm findet man unter *Graph/gentraceroute.py*. Die Ergebnisse sind in den Abbildungen 15-19 abgebildet.

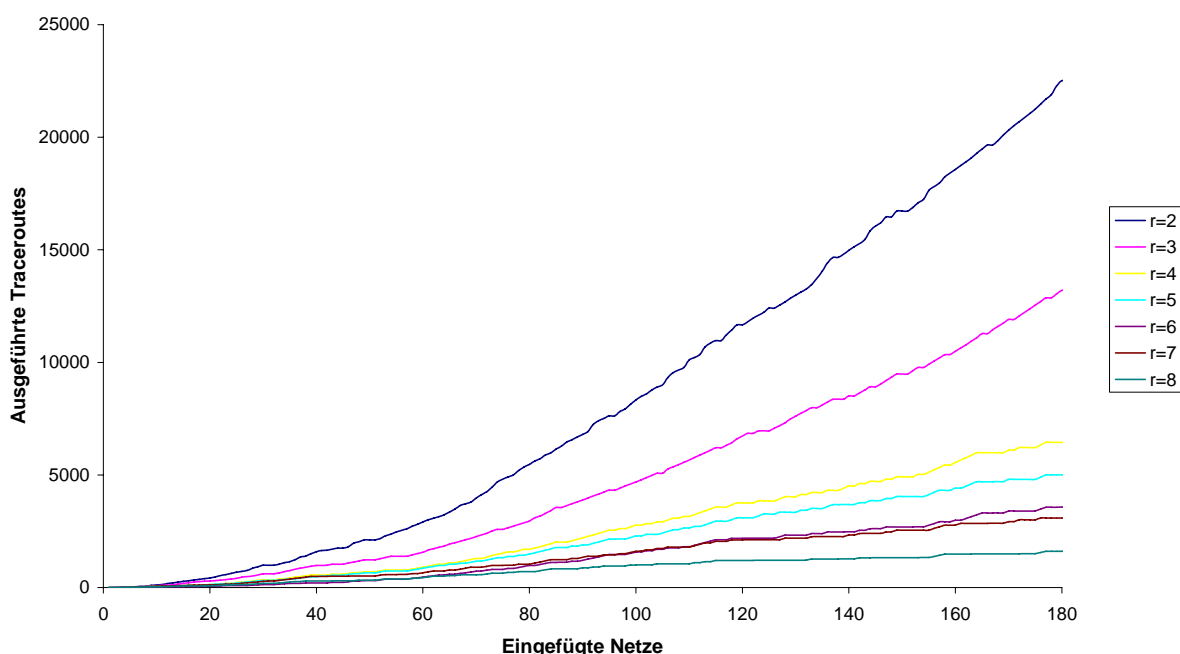


Abb. 15: Traceroutes pro Netz

Wie man in Abb. 15 als Ergebnis der Testläufe ganz klar sieht, kann der Traceroute-Aufwand um einen erheblichen Faktor durch die Wahl des Radius gesenkt werden.

Auch die Anzahl der Repräsentanten und somit die Größe der A-Matrix verringert sich, je höher der Radius gewählt wird, wie man in Abbildung 16 sieht. Jedoch steigt die Anzahl der Knoten pro Repräsentant logischerweise auch mit dem Radius (Abb. 17).

Nach der linearen Interpolation, die als Obergrenze dienen soll, kann man ausrechnen, bei welchem Radius wieviel Repräsentanten bei einer gewissen Netzanzahl entstehen werden. Bei der maximalen Anzahl, die der aus dem Caida-Graphen stammenden ca. 200000 Knoten, wäre man

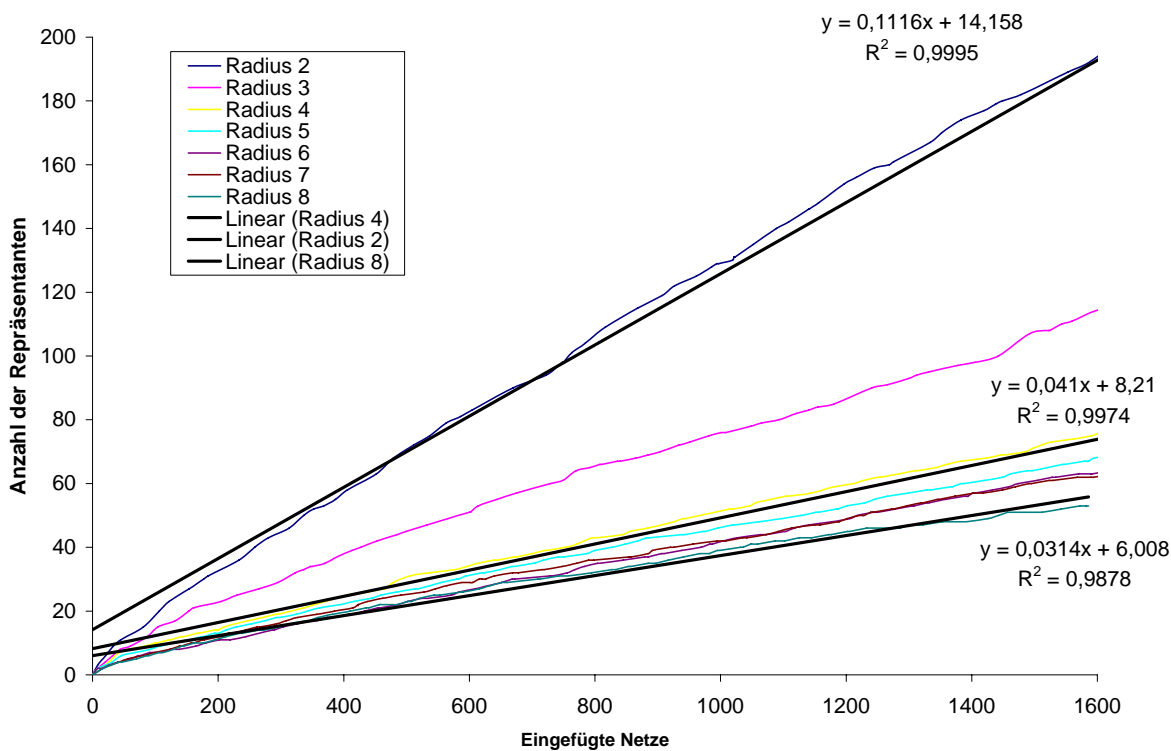


Abb. 16: Repräsentanten pro Netze

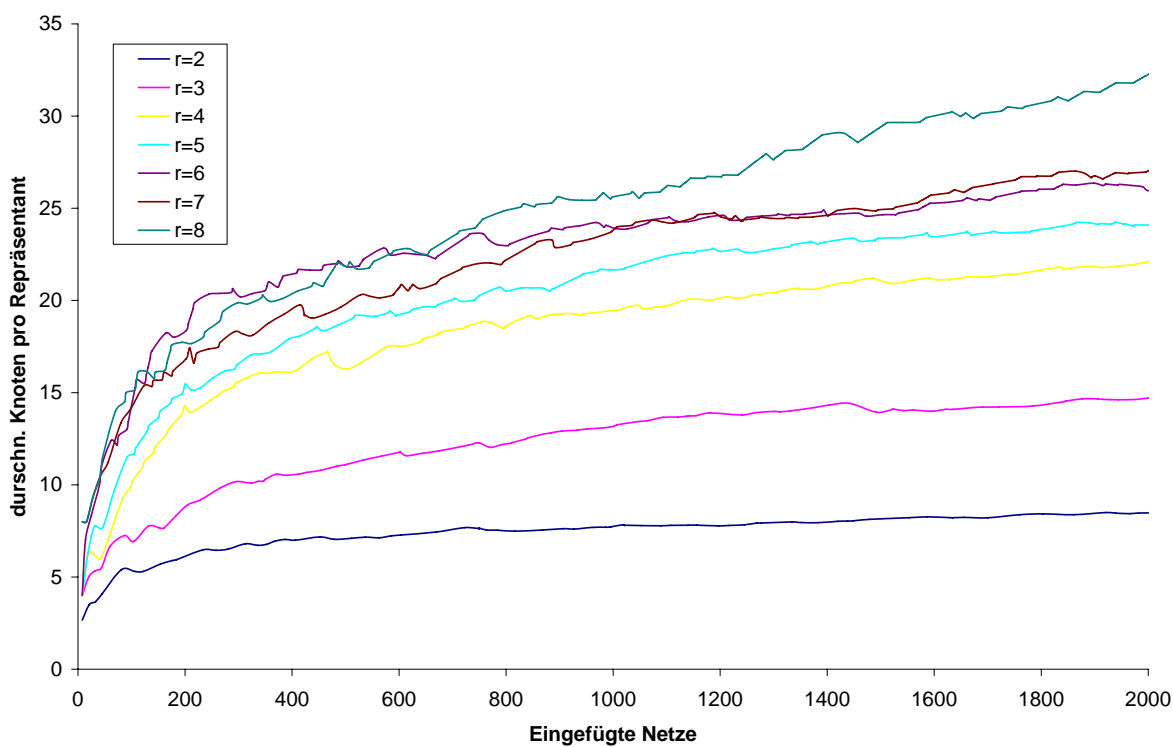


Abb. 17: Knoten pro Repräsentanten

bei einem Radius von 2 bei 22334 Repräsentanten, bei einem Radius von 4 bei 8208 und bei einem Radius 8 noch bei 6286 Repräsentanten. Das heißt bei einem Radius 4 und 200000 Netzen wäre die Größe der A-Matrix auf 0,17 Prozent der eigentlichen Größe geschrumpft.

#### 4.6.1 Überschneidungen

Überschneidungen können dadurch hervorgerufen werden, daß Kanten entdeckt werden, die einen Knoten doch näher zu einem anderen Repräsentanten erscheinen lassen, als zu dem, zu dem er vorher hinzugerechnet wurde.

Nun könnte man denken, daß es zu sehr vielen Überlappungen zwischen den einzelnen Bereichen kommen kann, welches natürlich den Speicherbedarf und die Effizienz des Algorithmus schmälern würde. Deshalb wurde während der Simulation mitprotokolliert, wieviele Bereichsüberschneidungen es gegeben hat, wobei bei Überschneidungen von mehreren Bereichen gleichzeitig, auch jede einzeln gezählt wurde.

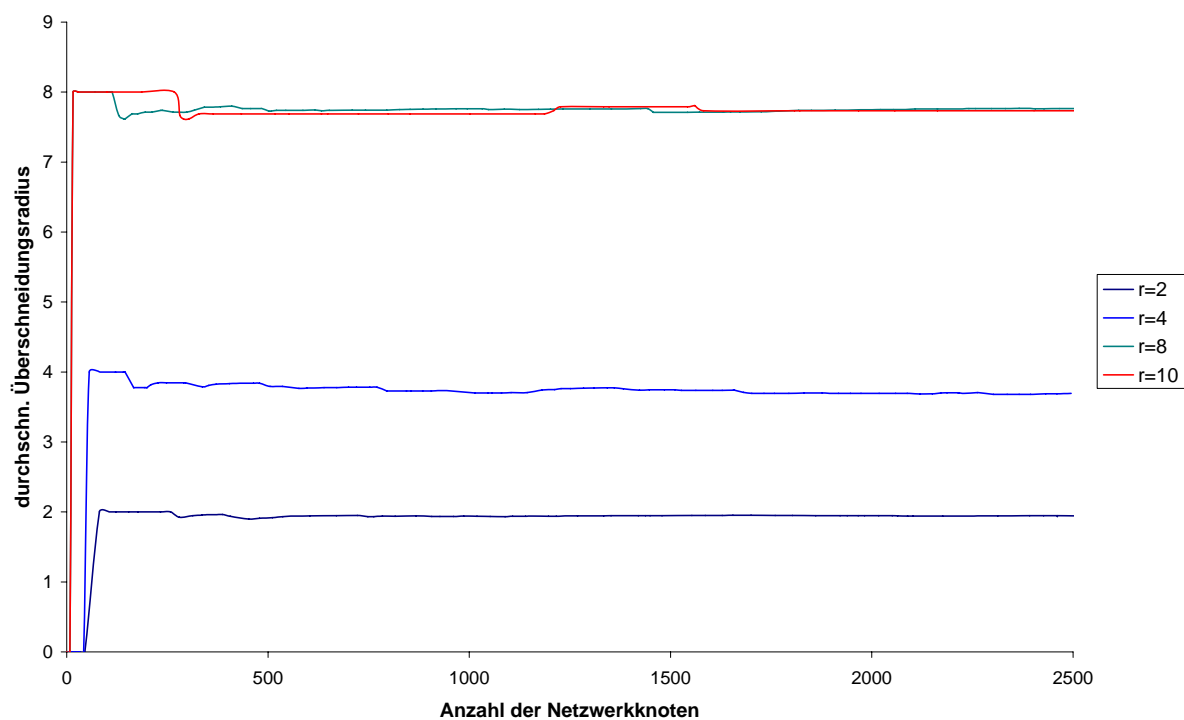


Abb. 18: Durchschnittliche Überschneidung von Repräsentanten

Wie man an dem Graphen aus Abb. 18 sehen kann, überschneiden sich die Bereiche im Durchschnitt nur knapp unter dem Radius der Repräsentanten. Über einem Radius von 8 (z.B. 10) gibt es, wie man sieht, keinerlei Überschneidungen mehr. Das liegt wahrscheinlich an der Natur des Graphen (siehe Abb. 7), dessen Nachbarschaftswachstum bei einem Radius von 8 sein Maximum überschritten hat.

Ein weiterer faszinierender Fakt ist, daß es bei einem Radius von 4 die wenigsten Überschneidungen gibt, wie in Abbildung 19 zu sehen ist, und die Anzahl der Überschneidungen auch am langsamsten wächst.

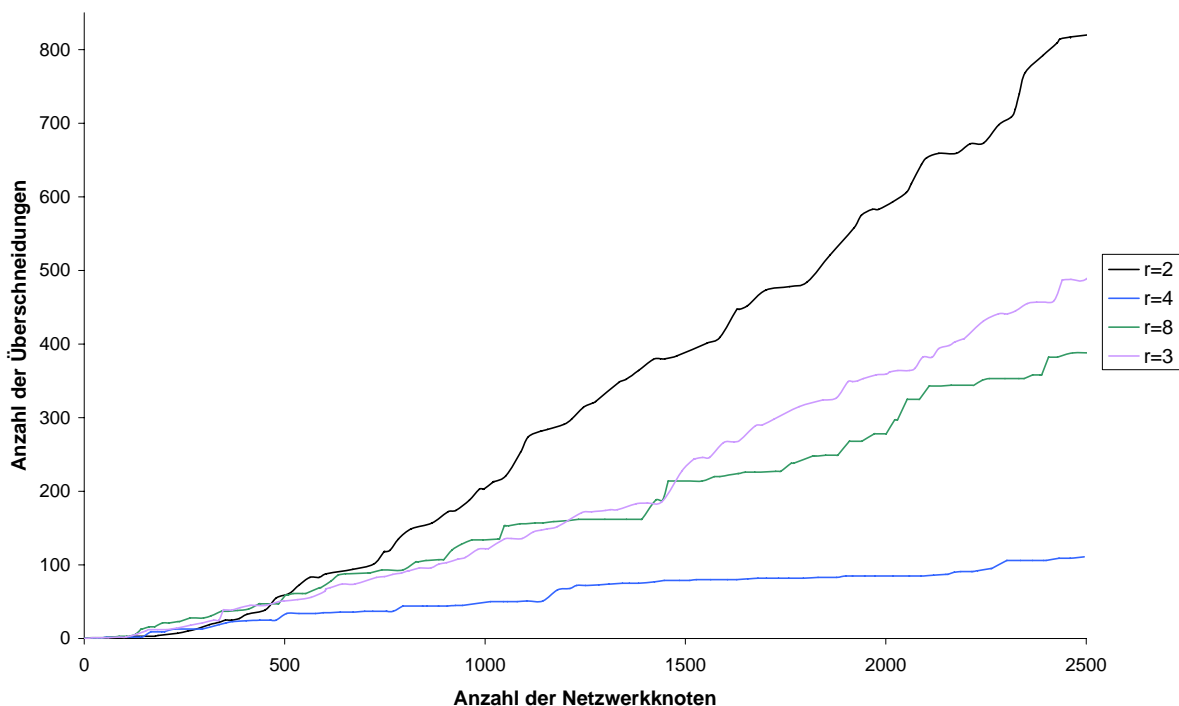


Abb. 19: Anzahl der Überschneidungen

## 4.7 Aufwandsabschätzungen für die A- und B-Berechnung

Eine Breitensuche kostet bei  $n$  Knoten und  $m$  Kanten  $O(m + n)$  [CLR97]. Wie in Kapitel 3.5 hinreichend erforscht wurde, liegt der durchschnittliche Knotengrad bei  $d = 6,62$  im Internet-Graphen. Die Anzahl der Kanten  $m$  in einem Graph mit einem Durchschnittsgrad  $d$  ist  $m = \frac{d \cdot n}{2}$ . Deshalb sind die Kosten einer Breitensuche in diesem Fall eingesetzt  $O(\frac{d \cdot n}{2} + n) = O(d \cdot n)$ , da  $d$  konstant  $O(n)$ . Das bedeutet die Grundkosten  $O(n)$  für die Breitensuchen.

Nach der Breitensuche muß für die Berechnung der Matrix jeder Knoten aus der einen Breitensuchen, mit den Knoten aus der zweiten Breitensuche kombiniert werden. Wenn jede Breitensuche alle  $n$  Knoten zurückliefert, dann kostet das  $O(n^2)$ .

Also liegen wir insgesamt bei  $O(n + n^2) \in O(n^2)$ . Da die Berechnung der A-Matrix für jede Kante im Graph einmal ausgeführt werden muß, liegt man bei  $n$  Knoten:  $m \cdot O(n + n^2) = O(\frac{d}{2} \cdot (n^2 + n^3)) \in O(n^3)$ .

Die B-Matrix muß nur für jeden Traceroute einmal berechnet werden und liegt somit in  $t \cdot O(n + n^2) = O(t \cdot (n + n^2)) \in O(t \cdot n^2)$  für  $t$  Traceroutes.

Somit ist der Algorithmus für die A-Matrix in sofern besser als der Floyd-Warshall-Algorithmus, daß er insgesamt eine Laufzeit von  $O(n^3)$  hat und nicht für jeden eingefügten Traceroute  $O(n^3)$  Berechnungen benötigt, also bei  $O(t \cdot n^3)$  liegt.

Der Algorithmus zur Berechnung der B-Matrix kommt sogar mit  $O(t \cdot n^2)$  Aufrufen aus und somit haben beide Algorithmen zusammen für ein konstantes  $d$  eine Gesamtlaufzeit von  $O(n^3 + t \cdot n^2) \in O(n^3)$ .

## 5 Das PeerNear-System

Im Mittelpunkt dieser Arbeit steht die Implementation eines Prototyps, welcher nicht selbst eine Peer-to-Peer-Struktur beinhaltet, sondern auf einer Client-Server-Struktur basiert. Das System besteht aus drei Elementen: einmal die Peers an sich, die Client- Vermittlerknoten und als Zentrum den Server. In Abbildung 20 ist eine grafische Darstellung des Peernear-Systems zu sehen, welche die verschiedenen Anwendungskonfigurationen und die Flexibilität des Systems verdeutlichen soll.

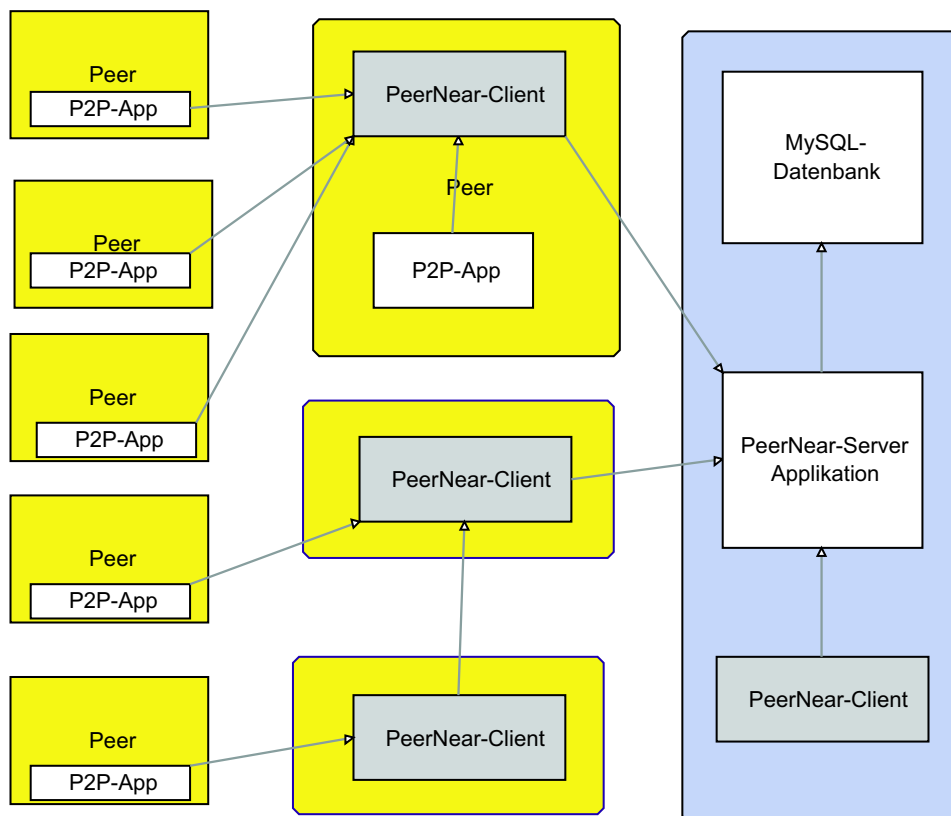


Abb. 20: Peer-Client-Server-Struktur

### 5.1 Server

Der Server ist für folgende Aufgaben zuständig:

- Sammlung und Vorhaltung der benötigten Betriebsdaten in einer Datenbank (Whois-Cache, etc.)
- Entgegennahme von Traceroute-Informationen und deren Verarbeitung
- Berechnung der A- und B-Matrizen

- Verteilung von Traceroute-Aufgaben an die Peers
- Nachrichtenweiterleitung von Peer-to-Peer

Auf die einzelnen Punkte und deren Realisierung im Server-Programm wird in Kapitel 6.1 noch weiter eingegangen.

## 5.2 Clients

Die Client-Instanzen sind eigentlich nur eine Art „Multiplexer“, damit möglichenfalls viele Peers an PeerNear teilnehmen können und nicht den Server mit TCP-Verbindungen belasten.

Clients verteilen im Prinzip nur adressierte Nachrichten vom Server auf die einzelnen Peers und umgekehrt. Das einzige was diese sich dabei merken müssen, ist die Abbildung der „virtuellen“ Adressen auf die IP-Adressen der Peers.

Rein theoretisch können mit dem TCP/IP-Protokoll  $2^{16}$  Verbindungen zu einem Rechner aufgebaut werden. Also können sich maximal  $2^{16}$  Clients mit einem Server und  $2^{16}$  Peers mit einem Client verbinden. Also insgesamt können sich maximal  $2^{32}$  Peers im Netzwerk aufhalten.

PeerNear-Clients könnten auch kaskadiert werden, wie in der Abbildung 20 zu sehen ist, damit eine noch größere Anzahl (unendlich viele) Peers ins Netzwerk aufgenommen werden können. Außerdem ist es auch möglich, einen Client auf dem gleichen Rechner wie den Server auszuführen.

## 5.3 Peers

Peers sind im Prinzip keine eigenständigen Peers, sondern bei PeerNear ein Teil der Peer-to-Peer-Applikation (also der „Client“ eines Peer-to-Peer-Netzes). Die Peer-to-Peer-Applikation baut eine Verbindung zu einem PeerNear-Client auf, wobei der Client auch auf dem gleichen Rechner laufen kann. Danach kann der Peer über den Client Informationen vom Server abfragen (z.B. wo sind meine nächsten Nachbarn), oder aber der Peer kann vom Server beauftragt werden, bestimmte Traceroutes durchzuführen, woraufhin der Peer die Ergebnisse zurückübermittelt.

## 5.4 Benötigte Software und Inbetriebnahme

Zur Ausführung des Server-Moduls ist ein Python-Runtime mit einer Version  $\geq 2.3.4$  (erhältlich unter <http://www.python.org>) notwendig, sowie die aktuellste Python-MySQL-API, welche

unter <http://sourceforge.net/projects/mysql-python> erhältlich ist. Für die Client-Applikation ist nur die oben genannte Python-Runtime notwendig.

Wie man im Anhang G sehen kann, existieren einige Konfigurationsdateien, die angepasst werden müssen, bevor PeerNear den Betrieb aufnehmen kann. Danach muss selbstverständlich erst der Server mithilfe des Skripts `startserver.sh` und danach der Client mit dem Skript `startclient.sh` gestartet werden.

Für die Webseite wird ein Webserver, vorzugsweise ein Apache (<http://www.apache.org>) benötigt und eine Installation der GraphViz-Software (<http://www.graphviz.org>). Der Apache-Server sollte so konfiguriert werden, daß er fähig ist PHP- und CGI-Skripte aus dem PeerNear-Verzeichnis auszuführen.

## 5.5 Die Webseite

Diploma Thesis by [Markus Scherschanski](#) - University of Paderborn  
Mentor: Asst. Prof. Dr. rer. nat. [Christian Schindelbauer](#)

Disconnect My Peer ID is 288

```
My IP-Address is:192.168.1.20
Connected to Client!
Doing Trace to 217.20.118.113
Traceroute resulted in: [192.168.1.50,'217.0.116.106','217.0.71.82','62.154.4
YOUR NEAREST PEERS:
PEERID, NETWORK, IPADDR, PORT, DISTANCE
83L, 7L, '84.140.222.229', 1582L, 0),
133L, 7L, '84.139.3.214', 62109L, 0),
134L, 7L, '84.139.3.214', 62112L, 0),
136L, 7L, '84.139.3.214', 62124L, 0),
137L, 7L, '84.139.3.214', 62130L, 0),
146L, 7L, '84.139.0.98', 35039L, 0)
```

Radius:  Request Nearest Peers

- [Introduction](#)
- [Latest News](#)
- [Testing-Java-Applet](#)
- [Applet-FAQ](#)
- [Execute from cmdline](#)
- [Current netgraph as](#)
  - [GIF](#)
  - [SVG](#)
  - [Info on SVG](#)
  - [DOT-File-Source for Graphviz](#)
- [List of connected peers](#)
- [List of whoisdata](#)
- [Graph as PDF:](#)
  - [Date 2.5.2005](#)
  - [Date 8.5.2005](#)
- [Software used with](#)
- [Program-Sources](#)
- [Papers and Publications](#)

Abb. 21: Webseite mit gestartetem Applet

### 5.5.1 Der Netzgraph

Nach jeder Änderung in der Verbindungsgraphmatrix, wird in das Verzeichnis von dem aus PeerNear gestartet worden ist, eine DOT-Datei geschrieben, welche von der Webseite aus über den Menüpunkt „DOT-File-Source for Graphviz“ auch heruntergeladen werden kann.

Daraus wird über das CGI-Skript „makemap.cgi“ mittels des Aufrufs von GraphViz [GN00] eine graphische Repräsentation des aktuellen Netzgraphen erzeugt.

Dabei werden zwei Versionen generiert:

- eine GIF-Datei, die direkt im Webbrowser angezeigt werden kann
- eine SVG [JF03]-Datei, für die ein spezielles Plugin notwendig ist

GraphViz erlaubt sogar den einzelnen Knoten URLs zuzuordnen, die dann als „clickable map“ oder in SVG direkt angeklickt werden können. Bei dem Netzgraph, führen die Links alle zur Datei *dosome.php*, die als Parameter die Knotennummer übergeben bekommt. Daraufhin wird der komplette Whois-Eintrag, der zuvor aus der MySQL-Datenbank (siehe dazu Anhang D) geholt wurde, angezeigt.

### 5.5.2 Andere wichtige Menüpunkte

- **list of connected peers** - startet ein PHP-Skript, welches aus der MySQL-Datenbank den Inhalt der Peerdaten abfragt. Somit sieht man welche Peers im Moment an dem Netzwerk teilnehmen.
- **list of whoisdata** - zeigt den Inhalt der Whois-Datentabelle an und die Liste, aller bisher hinterfragten Netze. Den letzten Stand der ersten Testphase sieht man im Anhang E.

## 5.6 Das Java-Test-Applet

Das Java-Applet soll die Betriebssystem- und Programmiersprachenunabhängigkeit demonstrieren. Ausserdem stellt es einen Pseudopeer eines Testnetzwerks dar, um die Funktionalität von PeerNear testen zu können.

### 5.6.1 Voraussetzungen

Prinzipiell kann das Test-Applet auf jedem Rechner ausgeführt werden, der eine Java Runtime Umgebung (JRE)  $\geq 1.4$  installieren und eine Form von Traceroute ausführen kann. Dabei ver-

sucht das Applet unter Windows „tracert“ auszuführen und unter Unix/MacOS-Betriebssystemen „traceroute“.

Bevor man das Applet von einem Webbrowser ausführen kann, muß man bestätigen, daß man dieses signierte Applet ausführen möchte. Dies ist nötig, damit das Applet fähig ist, das lokale Traceroute-Programm auszuführen.

### 5.6.2 Funktionsweise

Sobald das Applet gestartet wird, kann man durch klicken auf „Connect to Client“ eine Verbindung zu dem PeerNear-Client aufbauen. Dieser veranlasst das Applet einen Traceroute zu dem Server auszuführen, um den Peer einordnen zu können. Daraufhin schickt das Testapplet, das Ergebnis des gemachten Traceroute zurück zum Server. Dieser verarbeitet das Ergebnis und veranlasst ggf. andere Traceroutes. Sobald der Server einen Traceroute-Wunsch hat, kann er nun über das Applet den Peer „fernsteuern“ und beliebige Traceroutes veranlassen.

Falls man jetzt den Button „Request nearest Peers“ betätigt, bekommt man eine Liste aller Peers in dem Radius angezeigt, den man zuvor in dem Textfeld daneben angegeben hat. Dies ist in Abbildung 21 zu sehen.

Sobald man den Button „Disconnect“ drückt, wird die Verbindung zum Client abgebrochen.

### 5.6.3 Kommandozeilen-Version des Java-Applets

Die Kommandozeilen-Version des Java-Applets kann folgendermassen gestartet werden:

```
java -cp pnpeer.jar pnpeer.pnpeerappl
```

Danach sind folgende Eingaben möglich:

- **getnear (radius)** - stellt eine Anfrage nach allen Peers im Umkreis  $r$
- **msg (peerid) (msg)** - sendet eine userdefinierte Nachricht  $msg$  zu Peer  $peerid$ . Falls  $peerid$  0 angegeben wird, wird eine Nachricht an alle anderen Peers gesendet.
- **quit** - verlasse Programm

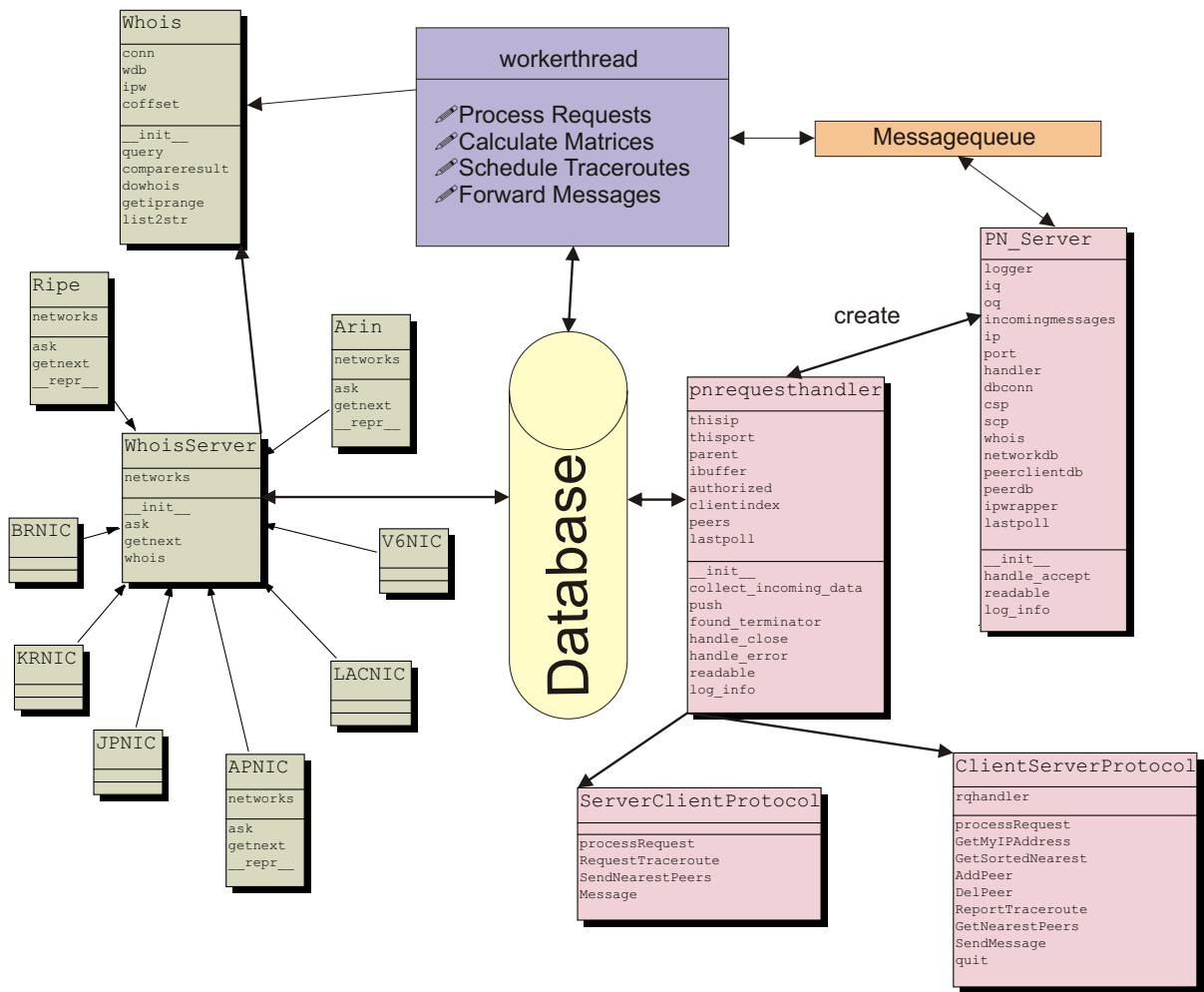


Abb. 22: Übersichtsdiagramm des Servermoduls

## 6 Implementation

Die Implementation des Client- und des Server-Systems wurde in der relativ neuen Skriptsprache Python vorgenommen. Einerseits hat Python ziemlich hochentwickelte Bibliotheken für Client-Server-Systeme und Netzwerkfunktionen, andererseits ist es für alle relevanten Zielplattformen (Windows, Unix, MacOS) erhältlich, da der Kern von Python (in C geschrieben), leicht portierbar ist. Auch das weltweit größte Peer-to-Peer-Netzwerk BitTorrent ist in Python geschrieben worden.

### 6.1 Servermodul

In Abbildung 22 ist eine kleine Übersichtsgraphik, über die Interaktionen und Abhängigkeiten innerhalb des Servermoduls zu sehen.

Das Servermodul besteht aus folgenden Hauptmodulen:

### 6.1.1 Asynchroner Socketserver

Datei: *Server/PNServer.py*

Der asynchrone Socketbetrieb hat gegenüber dem synchronen Socketbetrieb viele Vorteile und das vor allem, wenn sehr viele Clients auf TCP-Verbindungen unterhalten wollen.

Bei dem synchronen Socketbetrieb wird für jede Verbindung die neu geöffnet wird, ein neuer Prozess gestartet, der dafür zuständig ist, die ankommenden Daten entgegenzunehmen und zu verarbeiten. Allerdings sieht man gleich das Problem, die einzelnen Prozesse untereinander zu koordinieren, zu synchronisieren und eine Interprozesskommunikation zu etablieren. Außerdem belasten so viele Prozesse die Systemleistung erheblich und andauernde Kontextwechsel würden die komplette Performanz „auffressen“.

Da möglichst viele Clients bedient werden wollen, muß man die Sockets asynchron abfragen, d.h. wenn ein Client eine Verbindung öffnen will, wird diese etabliert und eine Instanz (in unserem Fall *pnrequesthandler*) angelegt, die die Daten dieser Verbindung verwaltet. Nun werden alle Verbindungen in einer Liste festgehalten und je nach Betriebssystem, entweder nacheinander (polling) oder ereignisgesteuert abgefragt, d.h. sobald Daten von irgendeiner Verbindung vorliegen, wird die entsprechende Instanz mit den Daten als Parameter aufgerufen.

Diese komplette Funktionalität stellt Python schon in seinem standartmäßigen *Asyncore*- bzw. *Asynchat*-Modul bereit.

### 6.1.2 Arbeitsprozess

Datei: *Server/PNServer.py*

Die Socketkommunikation geschieht also in einem einzelnen Prozess und somit dürfen die Methoden, die eine Anfrage entgegennehmen, nicht lange dauern, da während dieser Zeit die anderen Verbindungen keine Daten übermitteln können.

Deshalb wird ein zweiter Prozess, der „workerthread“ gestartet, der die Aufgaben erledigen soll, die längere Zeit in Anspruch nehmen können, und zwar:

- Datenbank-Abfragen
- Whois-Lookups
- Ausführen des Traceschedulers-Algorithmus (viele Matrix-Operationen)

- Verteilen der Traceroute-Aufträge an die Clients
- Anfragen nach nächsten Peers bearbeiten
- Peers ein- und austragen

Um zwischen dem Hauptprozess und dem Arbeitsprozess kommunizieren zu können, wurden zwei multitaskingsichere Nachrichtenwarteschlangen zwischengeschaltet, um eine reibungslose Kommunikation garantieren zu können.

### 6.1.3 Whois-Lookup

Dateien: *Server/whois.py* und *Server/whoisserver.py*

Die Whois-Klasse liefert im wesentlichen immer das zugehörige Netzwerk, welches bei dem entsprechenden RIR registriert ist. Zuerst wird eine Datenbank-Abfrage gestartet, ob das Subnetz zu dem die IP-Adresse zugehörig ist schonmal nachgefragt wurde und es auch sicher erscheint, daß dieses Subnetz keine weiteren Unterteilungen hat. Davon kann ausgegangen werden, wenn das Netz nicht mehr als 256 IP-Adressen umfasst. Alternativ könnten auch größere Intervalle angegeben werden, damit die Anzahl der Whois-Abfragen gesenkt werden kann. Als Beispiel umfasst das Netzwerk der Universität Paderborn ungefähr  $2^{16} = 65535$  mögliche IP-Adressen (minus die Broadcastadressen noch 65024).

Falls das Netzwerk noch nicht erfragt wurde, bzw. die Möglichkeit besteht, daß es noch weiter unterteilbar ist, wird eine Whois-Anfrage mit der IP-Adresse des Peers, ausgeführt.

Als erstes wird der Whois-Server von RIPE (Réseaux IP Européens) befragt, da sich ein Hauptteil der Anfragen auf europäische Netze beziehen wird. Dazu wird eine TCP-Verbindung auf Port 43 zu der Adresse `whois.ripe.net` aufgebaut und einfach die zu hinterfragende IP-Adresse im Klartext mit einem abschliessenden Zeilenvorschub (CRLF) gesendet. Daraufhin wird der komplette Whois-Eintrag zurückgesendet. Ein Beispielhaftes Exemplar sieht man im Anhang A.

Daraus werden nur die beiden Informationen Netzname und IP-Bereich, bei RIPE *inetnum*, *netname* extrahiert. Der Vollständigkeit halber und da diese Information automatisch mitgeliefert wird, wurde auch noch das Feld *origin* an den Namen des Netzes angehängt, welches die Nummer des Autonomen Systems enthält.

Für alle IP-Adressen die nicht bei RIPE verwaltet werden, gibt die Whois-Datenbank von RIPE einen Eintrag zurück, daß alle Adressen zwischen 0.0.0.0 und 255.255.255.255 zur IANA (Internet Assigned Numbers Authority) gehören. Daran kann man erkennen, daß man nun woanders nach dem Besitzer dieser IP-Nummer suchen muss.

Falls also die Suche bei RIPE nicht erfolgreich war, wird als nächstes ARIN gefragt (American Registry for Internet Numbers). Diese Datenbank scheint die gepflegteste zu sein und verfügt über ein großes Übersichtswissen, welcher IP-Bereich wohin gehört. Falls man dort nicht fündig wird, bekommt man über das Feld *ReferralServer* gesagt, welchen Whois-Server man fragen muss, um eine direkte Antwort zu bekommen (z.B. `whois://whois.apnic.net`). Da es keinen einheitlich Standard für Whois-Datenbankabfragen gibt, unterscheiden sich die relevanten Felder hier ein wenig: *inetnum* wird zu *NetRange*, *netname* wird zu *NetName*. Auch kann es verschachtelte Abfragen geben. Wenn mehrere „NetHandles“ bei der Abfrage zurückkommen, dann ist der IP-Bereich mehrfach unterteilt und man muss mehrere Whois-Abfrage mit den gefunden NetHandles machen, um an die korrekten Informationen zu gelangen.

Von ARIN aus werden nun nur noch Anfragen an den APNIC weitergeleitet, welcher ganz Asien umfasst. Es gibt noch viele kleine Whois-Server z.B. JPNIC oder LACNIC. JPNIC wird vom APNIC gespiegelt, und LACNIC wird seit 2002 auch von ARIN übernommen. Jedoch wurde für zukünftige Verfeinerungen der Whois-Abfragen die Möglichkeit gegeben, die Abfragemöglichkeiten für diese kleineren Registries, nachzurüsten.

Wenn der richtige Whois-Eintrag gefunden ist, werden abschliessend die gesammelten Daten in die Datenbank geschrieben. Dazu müssen nur noch die IP-Adressen von der „Dotted Decimal Notation“ in normale Integer umgewandelt werden, damit sie in der Datenbank nach IP-Adressen sortiert werden können.

Also wird aus den folgenden Zeilen im Whois-Eintrag:

```
inetnum: 131.234.0.0 - 131.234.255.255
netname: UNIPADERBORN
origin: AS1275
origin: AS680
```

diese Zeile in der Datenbank:

netindex	ipfrom	ipto	netname	hasubnets	gotanswer	rawtext
5	2213150720	2213216255	.UNIPADERBORN.AS680.AS1275.	0	Ripe	...

An das Hauptprogramm wird nun nur noch der Index des Netzes zurückgeliefert, welcher zu der übergebenen IP-Adresse gehört.

#### 6.1.4 Graph-Algorithmus

Dateien: *Graph/represent.py* und *Graph/graphapi.py*

Als Parameter kann man dem Traceroute-Scheduler beim Start des Servers die Größe des Radius  $r$  und die untere Schranke  $s$  (siehe Kapitel 4) zuteilen.

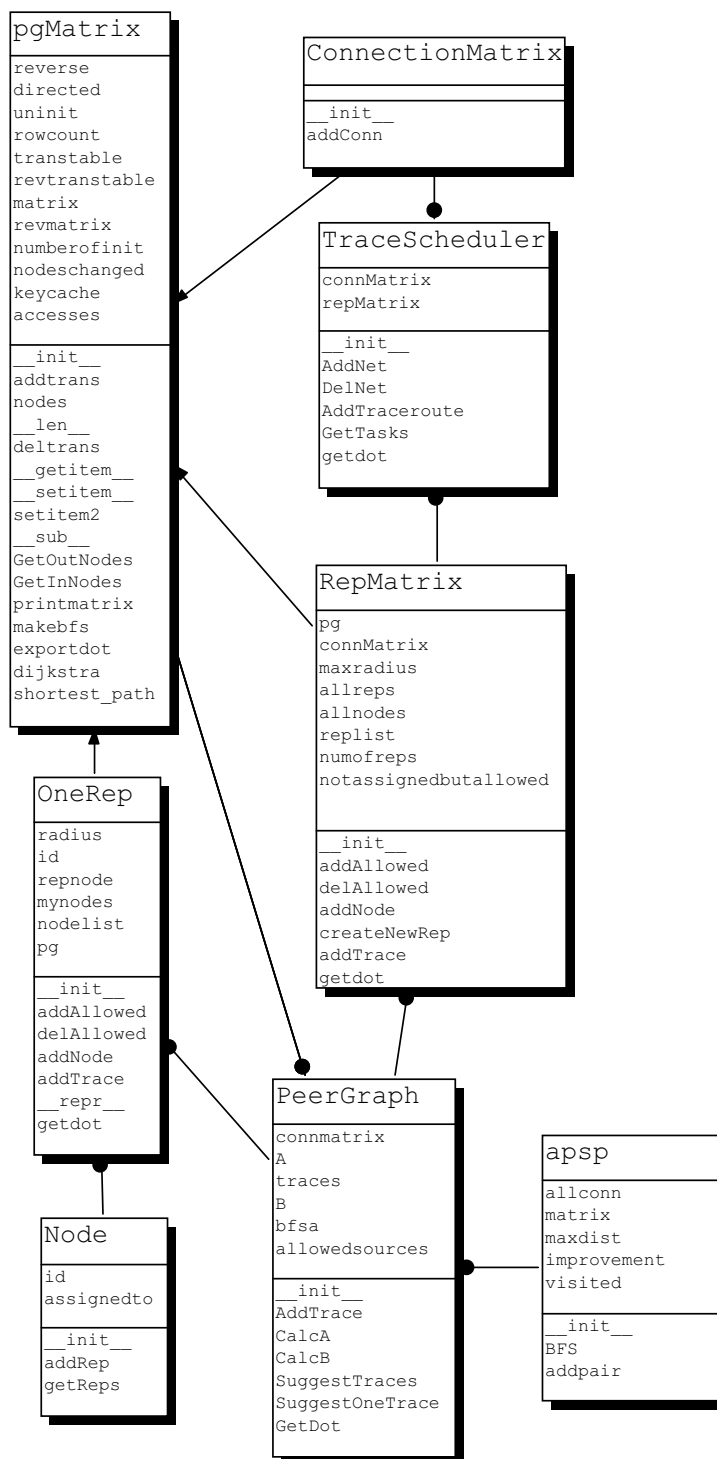


Abb. 23: Klassendiagramm Graph-Algorithmen

Nachdem die Indizes vom Whois-Modul zurückgeliefert wurden und der Traceroute nun anstatt IP-Adressen, Netzwerk-Indizes enthält, werden die Traceroutes sukzessive in den Traceroute-Scheduler mit der Funktion *AddTraceroute* eingefügt.

Damit der Traceroute-Scheduler weiß, in welchen Netzen tatsächlich Peers sitzen, die als potenzielle Aufrufer von Traceroute agieren könnten, gibt es die Funktion *AddNet* bzw. *DelNet*, die immer dann aufgerufen werden, wenn ein Peer hinzustößt oder die Verbindung zu ihm verloren geht. Mit *GetTasks* holt sich der workerthread die Vorschläge für die als nächstes lohnenden Traceroutes ab. Aus welchen Teilen der Graph-Algorithmus besteht, kann man in Abbildung 23 sehen.

**pgMatrix** ist eine extra für PeerNear angefertigte Matrixklasse, welche um den Speicherverbrauch für dünnbesiedelte Matrizen möglichst klein zu halten, nur die Daten speichert, welche von einem zu spezifizierenden Wert  $w$  (z.B. 0) abweichen, d.h. es werden nur die Daten gespeichert welche ungleich  $w$  sind. Für alle anderen nicht gespeicherten Paare  $ij$ , wird einfach der Wert  $w$  zurückgegeben. Die Datenstruktur dahinter ist ein zweidimensionales Dictionary, welches in Python auf einem dynamischen Hashtable basiert.

Außerdem stellt diese Matrix-Klasse noch Funktionen zur Verfügung um den Dijkstra-Shortest-Path-Algorithmus auf dem Graphen auszuführen und eine Methode um den Graph, den diese Matrix-Klasse repräsentiert, als DOT-Datei für GraphViz zu exportieren.

**PeerGraph** ist eine pgMatrix, worin ein beliebiger Graph gespeichert werden kann, worauf dann der Tracescheduler-Algorithmus angewendet wird. Er speichert die A- und B-Matrix. Man kann mit *AddTraceroute* einen Traceroute hinzufügen und über *SuggestTraces* die nächsten Vorschläge für Traceroutes erfragen.

**apsp** implementiert den All-Pair-Shortest-Path-Algorithmus, wie er schon im Kapitel 4 skizziert wurde und wird von der PeerGraph-Klasse aufgerufen um die A-Matrix zu berechnen.

**ConnectionMatrix** ist abgeleitet von pgMatrix und repräsentiert den Verbindungsgraph zwischen den einzelnen Netzen eins zu eins. Dort werden nur die jeweiligen Kanten der Traceroutes abgespeichert.

**RepMatrix** stellt die übergeordnete Repräsentantenebene dar, bzw. den Verbindungsgraph zwischen den Repräsentanten und benutzt die PeerGraph-Klasse um den Traceroute-Scheduler-Algorithmus darauf anzuwenden. Ein Knoten in diesem Graph wird durch eine Instanz OneRep-Klasse repräsentiert.

**Node** stellt einen Knoten, bzw. ein Netzwerk im Graphen dar und ist somit die kleinste Einheit. Mehrere Knoten werden zu einem Repräsentanten zusammengefasst in der OneRep-

Klasse

**OneRep** ist ein Repräsentant welcher aus mehreren Knoten bestehen kann, die von der Nodes-Klasse implementiert werden. Außerdem bildet ein ausgesuchter Knoten dieser Klasse das Zentrum des Repräsentanten bzw. ist der Repräsentant dieses Bereichs.

## 6.2 Clientmodul

Das Clientmodul, welches lediglich die Kommunikation zwischen Server und Peer regelt und als eine Art Verbindungsmultiplexer arbeitet, baut beim Start eine Verbindung zum Server auf (*pnserverconnection*) und hält ab da wieder einen asynchronen Socketserver (siehe 6.1.1) bereit (*PN\_Client*).

Sobald ein Peer eine Verbindung zu dem Client aufbaut, wird eine neue Instanz der Klasse *pnpeerconnection* aufgebaut, welche ab da diese Verbindung verwaltet und sobald eine Nachricht vom Peer eintrifft, diese an die Klasse *pnserverconnection* übergibt. Andersherum funktioniert dieses genauso.

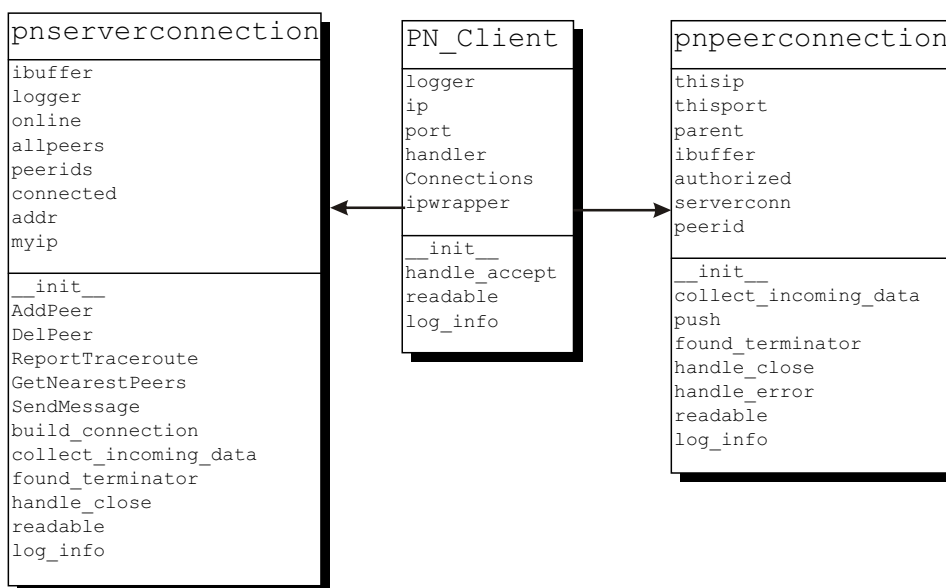


Abb. 24: Klassendiagramm Client-Modul

## 6.3 Java-Peer-Applet

Wie schon gesagt kann das Applet einmal als graphische Variante und einmal von der Kommandozeile ausgeführt werden. Um beides in einem Modul vereinigen zu können, gibt es einmal die Klasse *pnpeerappl* für die Kommandozeile und *pnpeermain* die ein Applet implementiert,

welches in Webbrowsern gestartet werden kann.

Das Programm besteht aus zwei Prozessen, einmal der Benutzeroberfläche und einem Hintergrund-Task der die Verbindung mit dem Client hält (*PeerClientConnection*). Die Traceroute-Anfragen, die der Server an jeden Peer stellen kann, werden durch diesen Task ausgeführt und das Ergebnis wieder zurückgesendet. Außerdem kann die Benutzeroberfläche durch zugreifen auf die Funktionen des Hintergrund-Task eigenständig Anfragen an den Server starten.

Die *TraceRoute*-Klasse kann den Traceroute-Befehl des jeweiligen Betriebssystems ausführen und das Ergebnis wieder einlesen. Diese Klasse, genauso wie *PeerInetAddress* und *ITSocket* habe ich dem NetGraph-Programm [Mor04] entlehnt. Die letzteren Klassen müssen existieren damit die *TraceRoute*-Klasse funktionieren kann.

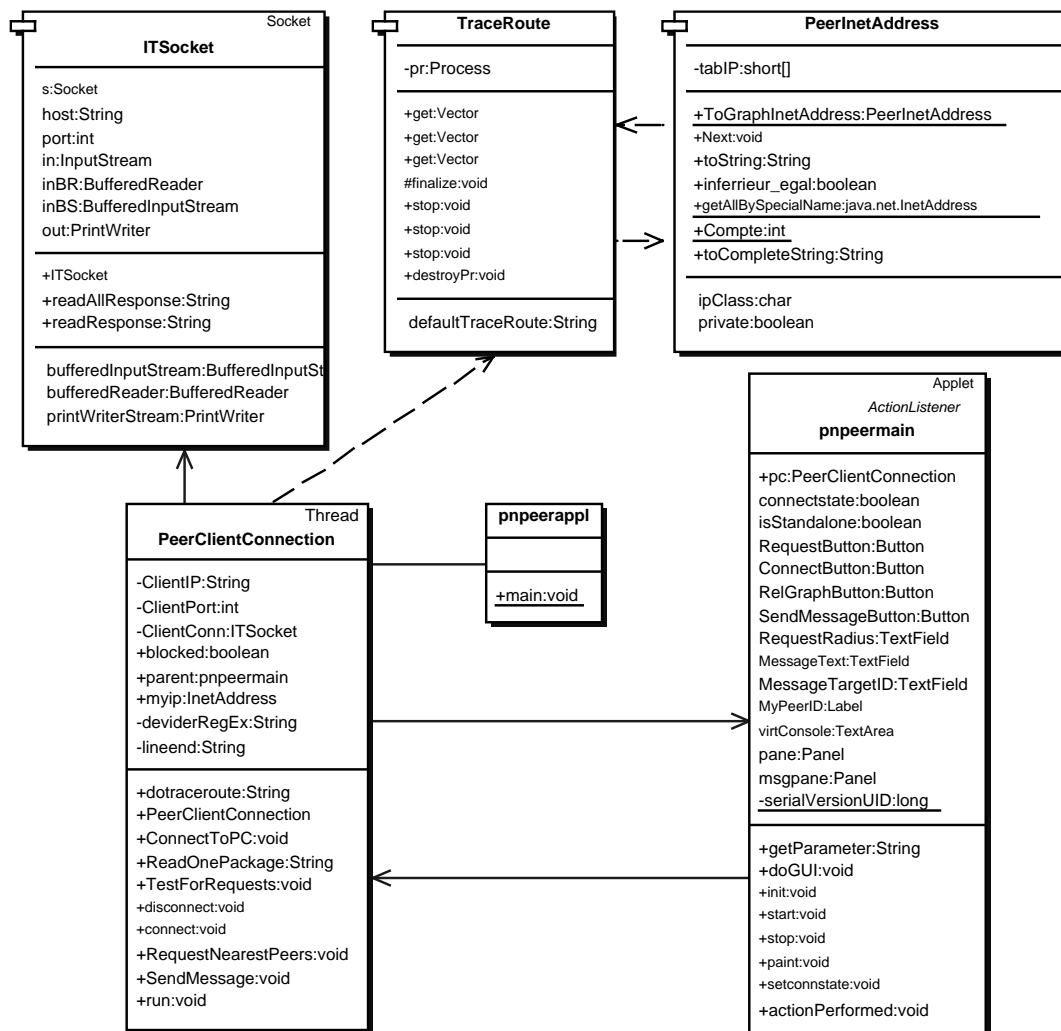


Abb. 25: Klassendiagramm Java-Peer-Applet

## 7 Zusammenfassung

PeerNear hat eine neue Gruppierungsform von IP-Bereichen eingeführt und zwar die IP-Registrar-Subnetz-Ebene, welche von der Auflösung zwischen einzelnen IP-Adressen und dem Zusammenfassen zu Autonomen Systemen steht. Dadurch ist der Aufwand nicht mehr abhängig von der Anzahl der zu verwaltenen Peers, sondern in wieviel verschiedenen Netzen diese sich aufhalten. Nach oben ist diese Anzahl der Netze durch deren Unterteilung im Internet begrenzt, welche ungefähr bei 200000 liegt, welches durch den Skitter-Graphen herausgefunden wurde. Diese Anzahl wird auch von der Anzahl der Autonomen Systemen in BGP-Systemen bestätigt [Hus05].

Die Natur des Graphen zwischen den Netzen wurde als ein Graph mit Pareto-Eigenschaften identifiziert. In dieser Arbeit wurde entdeckt, daß sich diese Pareto-Verteilungen innerhalb des Graphen überlagern können, so daß wenn man die Gradverteilung in große und kleine Grade aufteilt, auch eine genauere Funktion für diese Verteilung angeben kann. Außerdem wurde das Nachbarschaftszuwachstumsverhalten untersucht, welches sich von allen Punkten des Internets aus gleich verhält und somit dieses exponentielle Wachstum als Gesetzmäßigkeit für diesen Graphen angenommen werden kann. Dadurch konnte eine obere Schranke für den Durchmesser des Internet-Graphen auf Routerebene angegeben werden.

Um den Internet-Graphen mittels Traceroute-Aufrufen kartographieren zu können, wurde der Traceroute-Scheduler-Algorithmus entwickelt. Zuvor wurden einige Betrachtung über das Wesen und die Eigenschaften von Traceroute-Ergebnissen angestellt, wobei herausgefunden wurde, daß diese den Internet-Graphen wie Zeltstangen aufspannen. Deshalb wurde zu der Modellierung des Traceroute-Scheduler-Algorithmus diese Analogie verwandt, welche in einer komplett neuen Graphtheorie mündete, nämlich der Zeltheorie.

Der fertige Traceroute-Scheduler-Algorithmus ist nun in der Lage, durch zwei Parameter  $s$  (Schranke) und  $r$  (Radius) die Anzahl der Traceroute-Aufrufe zu reduzieren und trotzdem den Internetgraph mit stetig wachsender Genauigkeit bis zu einer Schranke  $s$  zu lernen.

Diese Berechnung wird auf einem zentralen Server vorgenommen. Damit aber überhaupt die Traceroutes von allen beteiligten Peers aufgerufen werden können, wurde ein Server-Client-Peer-System implementiert, welches die Peers untereinander und mit dem Server verbindet. Trotz der Client-Server-Eigenschaften skaliert es dabei sehr gut. Peers können sich dadurch jederzeit über die nächsten Peers in ihrem Netzwerk informieren.

Mit PeerNear können nun Peer-to-Peer-Netzwerk ihre netztopologische Lokalität, die in dieser Arbeit differenziert dargestellt wurde, verbessern und nach der Gesetzmäßigkeit des Small-World-Phänomens, denen der Internet-Graph folgt, alle anderen Lokalitäten gleich mit.

## 8 Ausblick

PeerNear soll in Zukunft seinen Platz in angewandten Peer-to-Peer-Netzwerken finden und für netztopologische Lokalität sorgen, mit all den Annehmlichkeiten die diese mit sich bringt, z.B. Senkung der Kosten für die Provider, Geschwindigkeitszuwachs, u.v.a. Über diese Entwicklung kann man sich jederzeit unter <http://www.peernear.net> informieren.

Das Potenzial, welches in PeerNear und den verwendeten Algorithmen steckt, kann natürlich noch weiter ausgebaut bzw. verbessert werden. Deshalb hier zum Abschluß ein paar Vorschläge, wie dies geschehen könnte.

### 8.1 Verbessertes Traceroute

Wie man gesehen hat, ist es sehr schwierig, wenn nicht unmöglich, direkt aus z. B. Java heraus an die untersten Netzwerkdienste zu kommen und es blieb als einzige Möglichkeit, der direkte Aufruf des Traceroute-Programms aus dem Java-Programm heraus.

Da diese Traceroute-Programme aber auch jeweils drei Roundtriptime-Messungen durchführen, die man für PeerNear nicht benötigt und den Traceroute um Faktor drei in die Länge ziehen, könnte man diese leicht einsparen. Traceroute für Unix in der Version 1.4a12 hat zum Beispiel eine Option -q dafür, aber da wir nicht davon ausgehen, daß jedes Traceroute diese Option hat, müsste man ein Programm schreiben, was zuerst austestet, welche Einstellungsmöglichkeiten das Systemtraceroute hat, oder man programmiert gleich ein neues.

Allerdings ist die direkte Manipulation der Netzwerk-Sockets auch von Betriebssystem zu Betriebssystem unterschiedlich und man muß Möglichkeiten finden, die Schnittstellen dazu, evtl. durch einen Wrapper der in C geschrieben ist, zu vereinheitlichen.

### 8.2 Dynamischer Graph

Falls man mit dem festen Radius für die Repräsentanten nicht einverstanden ist, könnte man die Radien auch dynamisch mit der Knotenanzahl wachsen lassen. Erste Überlegungen in diese Richtung haben jedoch gezeigt, daß es sehr aufwändig ist und wahrscheinlich die Vorteile, die dadurch entstehen, wieder zunichte machen würde. Jedoch könnte man durchaus weitere Untersuchungen in diese Richtung unternehmen.

Ein anderer Punkt ist, daß der Algorithmus im Moment noch nicht das Wegfallen von Verbindungskanten berücksichtigt, da die Verbindungskanten doch relativ statisch zu sein scheinen. Nur die Routen an sich können sich je nach Last verändern. Das Verschwinden von Verbindun-

gen kann nicht getestet werden, da man nicht weiß, ob diese Kante nun wirklich nicht mehr da ist, oder ob das Routing im Moment über sie nicht mehr erfolgt. Deshalb kann das Ausscheiden von Kanten nur durch eine Vergessensfunktion realisiert werden. Dabei muß evaluiert werden, ob es dadurch nicht vielleicht zu inkonsistenten Informationen im Algorithmus kommen kann, bzw. ob diese tragisch sind.

Realisiert werden kann die Vergessensfunktion dadurch, daß man sich merkt, wann diese Kante das letzte Mal benutzt wurde und wenn diese Zeit länger als ein gewisser Zeitraum  $t$  ist, wird sie gelöscht. Dieser Zeitraum müsste ziemlich lang sein, da es sein kann, daß für längere Zeit keine Erforschungen in diesem Bereich stattfinden. Wenn der Graph allerdings vollständig ist, also keine neuen Knoten mehr hinzukommen, verändert er sich gar nicht mehr und dann würden nacheinander alle Kanten aus dem Graphen gelöscht. Deshalb müsste man den Zeitraum  $t$  relativ zum Gesamtalter der umliegenden Kanten sehen.

Was der Algorithmus in dieser Version auch nicht berücksichtigt ist der Fluss, das heißt er vernachlässigt Bandbreite und Latenzzeit der Verbindungen. Also müßte getestet werden, wie dieser Algorithmus auf Kantengewichte ungleich 1 reagiert und ob er dann immer noch gilt. Außerdem müssten dann schnellere Verbindungen kleinere Gewichte haben als langsamere Verbindungen. Wenn man diese zwischen 0 und 1 antiproportional oder sogar exponentiell skaliert, könnte dies die Entscheidung zwischen zwei als gleichgut empfundenen Wahlmöglichkeiten (nach dem aktuellen Algorithmus) die Entscheidung bringen.

### 8.3 PeerNear als Peer-to-Peer-Netzwerk

Natürlich macht es am meisten Sinn, einen Dienst für Peer-to-Peer-Netzwerke auch als Peer-to-Peer-Netzwerk zu designen. Da nach dieser Diplomarbeit klar ist, welche Schritte benötigt werden und worauf es ankommt, dürfte dies nicht mehr zu schwer sein. Deswegen hier ein Versuch einer Skizzierung des zukünftigen PeerNear-Peer-to-Peer-Netzwerks:

Da man die Berechnung der Matrizen nur schlecht verteilen kann, dazu eine gemeinsame Speicherung der Matrizen bräuchte und der Overhead dafür die Performance auffressen würde, müsste jeder Peer selber die A,B,C,D-Matrizen verwalten und berechnen.

Was bliebe sind die Tabellen, die im Moment noch in der MySQL-Datenbank gehalten werden. Diese könnte man entweder über eine verteilte Datenspeicherung (z.B. auf einem Distributed Hash Table-Netzwerk) halten, oder auf jedem Peer diese Daten redundant verwalten und jeweilige Änderungen durch Broadcasts an alle anderen Peers im Netzwerk mitteilen.

Da die meisten Tabellen in der MySQL-Datenbank sowieso nur der Verwaltung der Client-Server-Struktur dienen, blieben nur noch die Whoisdaten und die einzelnen Peerdaten übrig.

Die Whoisdaten könnte man ganz einfach verwalten: wenn ein Peer ein Traceroute macht, dann versucht er gleich die erhaltenen IP-Adressen auf die Netzwerk-Indizes anhand seiner eigenen Whois-Cache-Tabelle abzubilden. Falls er dabei eine whois-Abfrage starten muss, fügt er das Netzwerk bei sich ein und teilt über Broadcast allen anderen Peers die Daten dieses neuen Netzwerks mit. Dazu muss noch nicht einmal der komplette Whois-Eintrag verschickt werden. Das einzige was hier beachtet werden muß ist, daß wenn zwei Peers gleichzeitig das neue Subnetz entdecken, das Subnetz nicht doppelt eingetragen wird und das die Indizes für die einzelnen Netzwerke nicht doppelt vergeben werden.

Nachdem der Traceroute auf die Netzwerk-IDs „gemapped“ wurde, wird der Traceroute auch über Broadcast für alle anderen Peers bekannt gemacht, woraufhin jeder Peer diesen Traceroute in seine A- und B-Matrix einsortiert. Anschliessend müssen die Peers auch nicht mehr die D Matrix bilden, sondern nur die Einträge von A-B berechnen, die sie selber betreffen. Daraus können sie dann schliessen, zu welchen Netzwerken Sie noch Traceroutes machen müssen.

Nun wäre nur noch zu klären, welche Daten über die Peers gespeichert werden müssten. Bei näherem Hinschauen sieht man, daß die Peers die anderen Peers garnicht kennen müssen. Es überprüft nur jeder, ob das Netzwerk, in dem er sich befindet, irgendwo hin einen Traceroute machen muss. Es gäbe noch die Möglichkeit, daß sich Peers, die sich im gleichen Netzwerk befinden, untereinander absprechen, wer dann immer den jeweiligen Traceroute ausführt. Dazu würde eine Broadcast-Nachricht ausreichen. Falls ein Peer neu dem Netzwerk beitrifft, teilt er mit, in welchem Netz er sich befindet und mit welcher IP-Adresse. Dann könnte man zur Regel machen, daß z.B. immer nur der mit derjenige mit der niedrigsten IP-Adresse die Traceroutes ausführt. Im Prinzip könnten sich die anderen in dem gleichen Subnetz auch das Vorhalten der Daten bzw. das Mitrechnen der Matrizen sparen.

Nun muß nur noch dafür gesorgt werden, daß falls ein neuer Peer dem Netzwerk beitrifft, er mit den bisherigen Matrizen-Ständen und Whois-Daten versorgt wird. Dazu schickt der Peer, mit dem sich der neue Peer verbunden hat, ein Willkommenspaket mit dem aktuellsten Stand der Daten.

## 8.4 PeerNear als Routing Protokoll

Wenn man sich PeerNear einmal genauer anschaut, dann wird man bemerken, daß es alle notwendigen Voraussetzungen für ein Routing-Protokoll, wie z.B. BGP oder OSPF besitzt und sich vielleicht sogar als besser geeignet herausstellen könnte.

Folgende äußerst günstigen Umstände sprechen dafür:

- PeerNear löst das All-Pair-Shortest-Problem in sehr guter Zeit und das als Update-Algo-

rithmus.

- Außerdem kann es intelligent und gezielt nach Verbindungen forschen, ohne jeweils die Umgebung mit Broadcasts zu überfluten.
- Durch das Gruppieren nach Radian, wird das Gruppieren von Routern zu Autonomen Systemen fast überflüssig. Die A-Matrix, die ja eigentlich ein quadratisches Wachstum aufweist, wird dadurch begrenzt.

Als Routing-Algorithmus könnte man sich folgende Vorgehensweise vorstellen: Wenn ein Datenpaket bei Knoten  $k_1$  ankommt und nach  $k_2$  geroutet werden soll, dann schau in der A-Matrix nach, welcher von den Knoten zu dem Knoten  $k_1$  eine Verbindung unterhält, den kleinsten Eintrag in der A-Matrix zu Knoten  $k_2$  hat, also den kürzesten Pfad. Natürlich wurde zuvor der Verbindungsgraph solange erforscht bis die B-Matrix nur noch gering von A abweicht. Dieser Grad kann natürlich während des Routings immer weiter verbessert werden.

Vielleicht könnten auch die Mechanismen von BGP benutzt werden, um AS-Pfade zu generieren, die dann als Traceroute-Ersatz in die A- und B-Matrix eingefügt werden könnten, welches dann eine Art Hybrid-Lösung bilden würde, damit BGP und ein neues Routing-System koexistieren könnten. Jedoch müssen all diese Möglichkeiten zukünftig noch verifiziert werden.

## Literaturverzeichnis

- [AR04] ARAUJO, F. ; RODRIGUES, L.: GeoPeer: A Location-Aware Peer-to-Peer System. In: *Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications (IEEE NCA04)*. Cambridge, MA, USA, August 2004, S. 39–46
- [Bor04] BORLAND, John: *Kazaa loses P2P Crown*. In: CNET News.com, Oktober 2004
- [CLR97] CORMEN, Thomas H. ; LEISERSON, E. ; RIVEST, Ronald L.: *Introduction to Algorithms*. MIT Press, 1997
- [DLS<sup>+</sup>04] DABEK, F. ; LI, J. ; SIT, E. ; ROBERTSON, J. ; KAASHOEK, M. ; MORRIS, R.: Designing a DHT for low latency and high throughput. In: *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*. San Francisco, California, March 2004
- [FFF99] FALOUTSOS, Michalis ; FALOUTSOS, Petros ; FALOUTSOS, Christos: On Power-law Relationships of the Internet Topology. In: *SIGCOMM '99*, 1999, S. 251–262
- [FJG04] FERREIRA, Ronaldo A. ; JAGANNATHAN, Suresh ; GRAMA, Ananth: Plethora: A Locality Enhancing Peer-to-Peer Network. In: *IEEE International Conference on Parallel and Distributed Systems*, 2004
- [GN00] GANSNER, E.R. ; NORTH, S.C.: *An open graph visualization system and its applications to software engineering (GraphViz)*. <http://www.research.att.com/sw/tools/graphviz/GN99.pdf>. Stand: 2000
- [HBc03] HYUN, Young ; BROIDO, Andre ; CLAFFY kc: Traceroute and BGP AS Path Incongruities / CAIDA Outreach. 2003. – Forschungsbericht
- [HHX<sup>+</sup>03] HEFEEDA, M. ; HABIB, A. ; XU, D. ; BHARGAVA, B. ; BOTEV, B.: Collectcast: A peer-to-peer service for media streaming. In: *ACM/Springer Multimedia Systems Journal* (2003), Oktober
- [HJS<sup>+</sup>03] HARVEY, Nicholas ; JONES, Michael B. ; SAROIU, Stefan ; THEIMER, Marvin ; WOLMAN, Alec: Skipnet: A scalable overlay network with practical locality properties. In: *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*. Seattle, WA, March 2003
- [HPMc02] HUFFAKER, B. ; PLUMMER, D. ; MOORE, D. ; CLAFFY k: Topology discovery by active probing (Skitter). In: *Proceedings of the Symposium on Applications and the Internet*. Nara, Japan, Januar 2002

- [Hus05] HUSTON, Geoff: *BGP Report for AS1221*. Webseite. <http://bgp.potaroo.net>. Stand: Juni 2005
- [IAN05] IANA: *INTERNET PROTOCOL V4 ADDRESS SPACE*. Webseite. <http://www.iana.org/assignments/ipv4-address-space>. Stand: Juni 2005
- [JB02] JIN, S. ; BESTAVROS, A.: Small-world internet topologies: Possible causes and implications on scalability of end-system multicast / Boston University. 2002 (4). – Forschungsbericht
- [JF03] JON FERRAILOLO, Dean J.: *Scalable Vector Graphics 1.1 Specification*. Webseite. <http://www.w3.org/TR/SVG/>. Stand: Januar 2003
- [Jol05] JOLTID: *PeerCache*. Webseite. <http://www.joltid.com/index.php/peer-cache/>. Stand: Juni 2005
- [JRT04] JAISWAL, S. ; ROSENBERG, A. ; TOWSLEY, D.: Comparing the structure of power-law graphs and the Internet AS graph / CS Dept., UMass Amherst. 2004 (04-30). – Forschungsbericht
- [Koe04] KOESTER, Jan-Paul: *ISP-proxy-caching for eMule (Project Homepage: WebCache eMule)*. Webseite. <http://webcache-emule.sourceforge.net>. Stand: 2004
- [MKKB01] MORRIS, Robert ; KARGER, David ; KAASHOEK, Frans ; BALAKRISHNAN, Hari: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In: *ACM SIGCOMM 2001*. San Diego, CA, September 2001
- [Mor04] MORVAN, Cyrille: *NetGraph: Traceroute graphique en Java*. Webseite. <http://cyrille.morvan.free.fr/netgraph/en/>. Stand: April 2004
- [Par04] PARKER, Andrew: *The True Picture of Filesharing*. Webseite. <http://www.cachelogic.com/research/slide1.php>. Stand: 2004
- [PF01] P. FRANCIS, C. Jin Y. Jin D. Raz Y. Shavitt L. Z.: IDMaps: A Global Internet Host Distance Estimation Service. In: *IEEE/ACM Trans. on Networking* 5 (2001), October, Nr. 9, S. 525–540
- [Pri04] PRIMETRICA: Global Internet Geography Report. Stand: 2004. <http://www.primetrica.com>. Primetrica. – Forschungsbericht. – Online-Ressource

- [Sch05] SCHÄFER, Volker: *DSL: Lohnen sich Billig-Flatrates für die Anbieter?* Online-Magazin. <http://www.teltarif.de/arch/2005/kw22/s17338.html>. Stand: Juni 2005
- [Sha04] SHAVITT, Y.: *Distributed Internet MEasurements and Simulations (DIMES)*. Webseite. <http://www.netdimes.org>. Stand: 2004
- [SMZ03] SRIPANIDKULCHAI, K. ; MAGGS, B. ; ZHANG, H.: Efficient Content Location Using Interest-Based Locality in Peer-to-Peer Systems. In: *INFOCOM*, 2003
- [Tan02] TANENBAUM, Andrew: *Computer Networks*. Prentice Hall Professional Technical Reference, 2002. – ISBN 0130661023
- [TK04] THOMAS KARAGIANNIS, Nevil Brownlee Kimberly C. Claffy Michalis F.: Is P2P dying or just hiding? / CAIDA Outreach. 2004. – Forschungsbericht
- [Wat99] WATTS, Duncan J.: *Small worlds: the dynamics of networks between order and randomness*. Princeton, NJ, USA : Princeton University Press, 1999. – ISBN 0-691-00541-9

## RFCs (Requests for Comments)

- [1] HEDRICK, C.: Routing Information Protocol / Internet Engineering Task Force. Version: Juni 1988. <http://www.rfc-editor.org/rfc/rfc1058.txt> (1058). – RFC. – Elektronische Ressource. – 33 S
- [2] MALKIN, G.: RIP Version 2 / Internet Engineering Task Force. Version: November 1998. <http://www.rfc-editor.org/rfc/rfc2453.txt> (2453). – RFC. – Elektronische Ressource. – 39 S
- [3] MOY, J.: OSPF Version 2 / Internet Engineering Task Force. Version: April 1998. <http://www.rfc-editor.org/rfc/rfc2328.txt> (2328). – RFC. – Elektronische Ressource. – 244 S
- [4] MOY, J.: OSPF Version 2 / Internet Engineering Task Force. Version: Juli 1991. <http://www.rfc-editor.org/rfc/rfc1247.txt> (1247). – RFC. – Elektronische Ressource. – 189 S
- [5] LOUGHEED, K. ; REKHTER, Y.: Border Gateway Protocol (BGP) / Internet Engineering Task Force. Version: Juni 1990. <http://www.rfc-editor.org/rfc/rfc1163.txt> (1163). – RFC. – Elektronische Ressource. – 29 S
- [6] HONIG, J. C. ; KATZ, D. ; MATHIS, M. ; REKHTER, Y. ; YU, Jiunn-Hwa.: Application of the Border Gateway Protocol in the Internet / Internet Engineering Task Force. Version: Juni 1990. <http://www.rfc-editor.org/rfc/rfc1164.txt> (1164). – RFC. – Elektronische Ressource. – 23 S
- [7] REKHTER, Y. ; LI, T.: A Border Gateway Protocol 4 (BGP-4) / Internet Engineering Task Force. Version: März 1995. <http://www.rfc-editor.org/rfc/rfc1771.txt> (1771). – RFC. – Elektronische Ressource. – 57 S
- [8] POSTEL, John: Internet Control Message Protocol / Internet Engineering Task Force. Version: September 1981. <http://www.rfc-editor.org/rfc/rfc792.txt> (792). – RFC. – Elektronische Ressource
- [9] REKHTER, Y. ; MOSKOWITZ, B. ; KARREBERG, D. ; DE, G. J. ; LEAR, E.: Address Allocation for Private Internets / Internet Engineering Task Force. Version: Februar 1996. <http://www.rfc-editor.org/rfc/rfc1918.txt> (1918). – RFC. – Elektronische Ressource. – 9 S

- 
- [10] SRISURESH, P. ; EGEVANG, K.: Traditional IP Network Address Translator (Traditional NAT) / Internet Engineering Task Force. Version: Januar 2001. <http://www.rfc-editor.org/rfc/rfc3022.txt> (3022). – RFC. – Elektronische Ressource. – 16 S
- [11] REKHTER, Y. ; LI, T.: An Architecture for IP Address Allocation with CIDR / Internet Engineering Task Force. Version: September 1993. <http://www.rfc-editor.org/rfc/rfc1518.txt> (1518). – RFC. – Elektronische Ressource. – 27 S
- [12] FULLER, V. ; LI, T. ; YU, J. ; VARADHAN, K.: Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy / Internet Engineering Task Force. Version: September 1993. <http://www.rfc-editor.org/rfc/rfc1519.txt> (1519). – RFC. – Elektronische Ressource. – 24 S
- [13] DAIGLE, L.: WHOIS Protocol Specification / Internet Engineering Task Force. Version: September 2004. <http://www.rfc-editor.org/rfc/rfc3912.txt> (3912). – RFC. – Elektronische Ressource

## Anhang A Whois-Eintrag der Uni-Paderborn

inetnum: 131.234.0.0 - 131.234.255.255  
netname: UNIPADERBORN  
descr: Universitaet Paderborn  
country: DE  
admin-c: GO209-RIPE  
tech-c: BB1-RIPE  
status: ASSIGNED PI  
mnt-by: DFN-LIR-MNT  
mnt-irt: IRT-DFN-CERT  
source: RIPE

person: Gudrun Oevel  
address: Universitaet Paderborn  
address: ZIT  
address: Warburger Strasse 100  
address: 33098 Paderborn  
phone: +49 5251 60 2397  
fax-no: +49 5251 60 4206  
e-mail: gudrun@uni-paderborn.de  
nic-hdl: GO209-RIPE  
mnt-by: DFN-HM-MNT  
source: RIPE

person: Barbara Bajer  
address: Universitaet Paderborn  
address: ZIT  
address: Warburger Strasse 100  
address: 33098 Paderborn  
address: Germany  
phone: +49 5251 60 5266  
fax-no: +49 5251 60 4206  
e-mail: barbara.bajer@uni-paderborn.de  
nic-hdl: BB1-RIPE  
mnt-by: DFN-NTFY  
source: RIPE

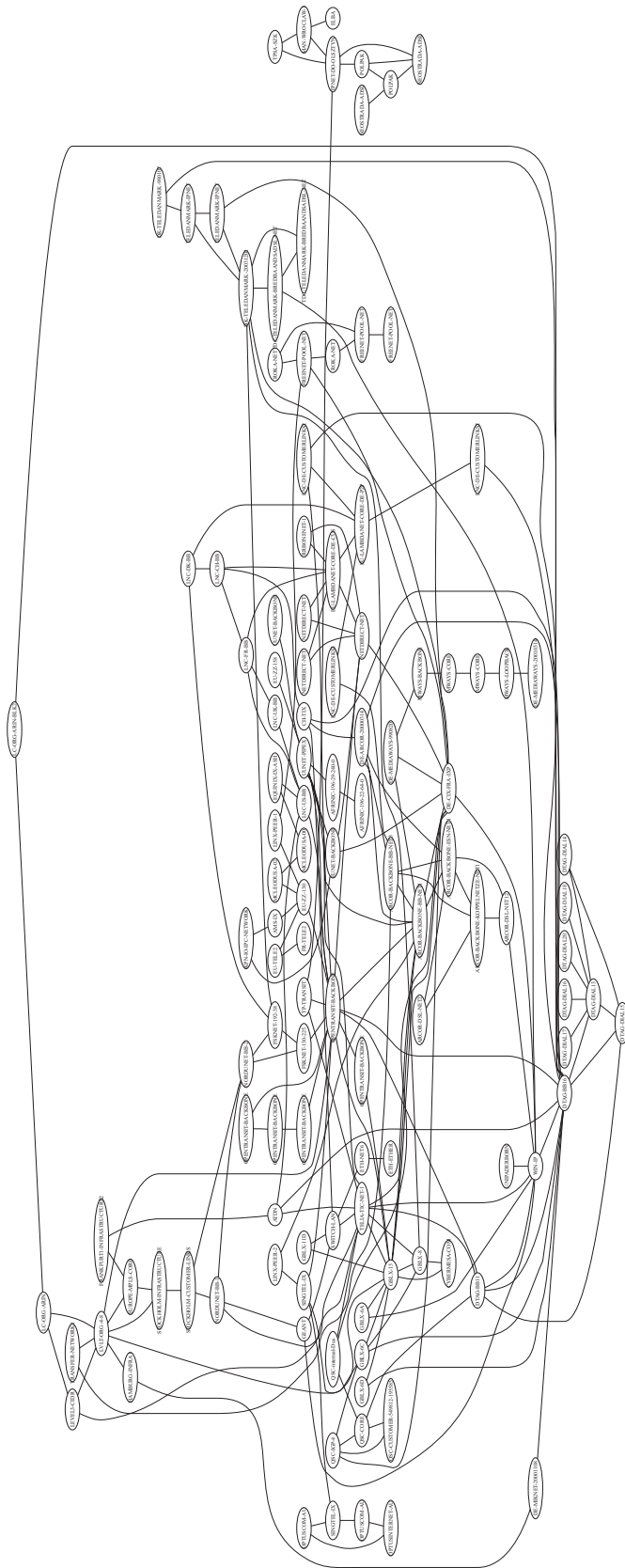
% Information related to '131.234.0.0/16AS1275'

route: 131.234.0.0/16  
descr: UNIPADERBORN  
origin: AS1275  
mnt-by: DFN-MNT  
source: RIPE

% Information related to '131.234.0.0/16AS680'

route: 131.234.0.0/16  
descr: UNIPADERBORN  
origin: AS680  
mnt-by: DFN-MNT  
source: RIPE

# Anhang B PeerNear-Graph (vom 6.6.2005)



# Anhang C Protokolle

## Anhang C.1 Peer über Client zu Server

- GetMyIPAddress()

Hiermit kann der Client seine eigene IP-Adresse erfragen, da Rechner hinter Firewalls nicht ihre eigene IP-Adresse herausfinden können

**Rückgabe:** IP-Adresse des Client

- AddPeer(IP-Adresse, PortAdresse, NetzName, NetzPasswort)

Wenn sich ein Peer bei einem Client anmeldet, meldet der Client ihn hiermit beim Server an

**Parameter:**

- IP-Adresse - *IP-Adresse des Peers*
- PortAdresse - *Portadresse von dem der Peer aus den Client kontaktiert hat*
- NetzName - *Name des Netzes in dem sich der Peer verbindet*
- NetzPasswort - *Authorisation für das Netzwerk*
- ExtraDaten - *Platzhalter für spätere Verwendungen*

**Rückgabe:**

- RejectPeer(IP-Nummer, PortNummer) falls der Peer abgewiesen wird, wegen falscher Authorisierung
- SetPeerID(Neue PeerID, IP-Nummer, PortAdresse) falls der Peer akzeptiert wird

- DelPeer(PeerID)

Client meldet damit einen Peer beim Server ab, falls der die Verbindung zum Client geschlossen hat.

**Parameter:**

- PeerID - *Index des Peers*

- ReportTraceroute(PeerID, VonIP, ZuIP, Traceroute-Liste)

Hiermit melden die einzelnen Peers die Resultate der Traceroute-Aufrufe an den Server zurück.

**Parameter:**

- PeerID - *Index des Peers, von dem der Traceroute stammt*
- VonIP - *IP-Adresse von der der Traceroute ausging*
- ZuIP - *IP-Adresse wohin der Traceroute ging*
- Traceroute-Liste *Liste von IP-Adressen, die das Ergebnis des Traceroute-Aufrufs waren*

- GetNearestPeers(PeerID, Radius)

Mit dieser Anfrage kann sich ein Peer eine Liste von Peers in einem gewissen Radius um sich herum, zurückgeben lassen.

**Parameter:**

- PeerID - *Index des Peers, von dem diese Nachricht kommt*
- Radius - *Radius auf dem die Suche nach benachbarten Peers beschränkt wird*

**Rückgabe:** siehe SendNearestPeers

- SendMessage(VonPeerID, ZuPeerID, Nachrichtentext)

Sende Benutzer-Nachricht von Peer zu Peer

**Parameter:**

- VonPeerID - *Index des Peers, von dem diese Nachricht kommt*

- ZuPeerID - *Index des Peers, an den diese Nachricht gerichtet ist*
- Nachrichtentext - *Benutzerdefinierter Inhalt der Nachricht*

## Anhang C.2 Server über Client zu Peer

- RequestTraceroute(PeerID, ZielIP)

Sende Aufforderung zur Ausführung eines Traceroutes

**Parameter:**

- PeerID - *Index des Peers, der den Traceroute aufrufen soll*
  - ZielIP - *IP-Adresse die als Ziel des Traceroutes angegeben werden soll*
- SendNearestPeers(PeerID, Liste der nächsten Peers)
- Sende Liste mit den nächsten Nachbarn, sortiert nach Entfernung, an Peer zurück.
- Parameter:**
- PeerID - *Index des Peers, der zuvor die Anfrage gestellt hat*
  - Liste der nächsten Peers - *Liste von Tuplen von Peers:  
(PeerID, Netzwerk, IP, Port, Entfernung)*
- Message(VonPeerID, ZuPeerID, Nachricht)

Versende eine benutzerdefinierte Nachricht an Peer über den Client

**Parameter:**

- VonPeerID - *Index des Peers, von dem die Nachricht stammt*
- ZuPeerID - *Index des Peers, an dem die Nachricht stammt*
- Nachricht - *Inhalt der benutzerdefinierten Nachricht*

## Anhang D Datenbank-Tabellen

(a) Whois-Daten

Feld	Typ	Kommentar
netindex	bigint(20)	Laufender Index der einzelnen Subnetze
ipfrom	int(10)	IP-Adresse ab der dieses Subnetz beginnt
ipto	int(10)	IP-Adresse mit der dieses Subnetz endet
netname	tinytext	Bezeichnung dieses Netzes
hassubnets	tinyint(1)	Ist dieses Netz unterteilt in weitere Netze
gotanswer	varchar(8)	RIR von dem diese Eintrag generiert wurde
rawtext	text	Rohfassung der Whois-Abfrage

(b) Beispiel Whois-Daten

netindex	ipfrom	ipto	netname	hassubnets	gotanswer	rawtext
1	3655335936	3656229167	.DTAG-DIAL15.AS3320.	0	Ripe	...

(c) Peer-Daten

Feld	Typ	Kommentar
peerindex	bigint(20)	Laufender Index der Peers
ipaddr	int(10)	IP-Adresse des Peers
port	int(10)	Port von der die Verbindung ausging
peernet	bigint(20)	Welchem Whois-Netzwerk ist der Peer angehörig (Index)
pnclient	bigint(20)	An welchem Client hängt dieser Peer (Index)
networkid	bigint(20)	Welchem Peer-to-Peer-Netzwerk ist er angehörig (Index)
lastupdate	timestamp(14)	Wann wurde dieser Eintrag zum letzten mal „geupdated“
extradata	text	Extraplatz für spätere Verwendung

(d) Beispiel Peer-Daten

peerindex	ipaddr	port	peernet	pnclient	networkid	lastupdate	extradata
26	2213167238	49981	5	1	1	20050429141331	

(e) Client-Daten

<b>Feld</b>	<b>Typ</b>	<b>Kommentar</b>
peerindex	bigint(20)	Laufender Index der Clients
ipaddr	int(10)	IP-Adresse des Clients
port	int(10)	Portadresse von der sich der Client aus verbunden hat
lastupdate	timestamp(14)	Zeitstempel für die letzte Änderung an diesem Eintrag
extradata	text	Extraplatz für zukünftige Anwendungen

(f) Beispiel Client-Daten

<b>peerindex</b>	<b>ipaddr</b>	<b>port</b>	<b>lastupdate</b>	<b>extradata</b>
1	2130706433	39068	20050429003015	

(g) Netzwerk-Daten

<b>Feld</b>	<b>Typ</b>	<b>Kommentar</b>
netindex	bigint(20)	Laufender Index der Peer-to-Peer-Netzwerke
netname	tinytext	Name des Netzwerks
netpasswd	varchar(12)	Passwort des Netzes
extradata	text	Extraplatz für zukünftige Anwendungen

(h) Beispiel Netzwerk-Daten

<b>netindex</b>	<b>netname</b>	<b>netpasswd</b>	<b>extradata</b>
1	testnet	testpasswd	

## Anhang E Whois-Cache-Daten (vom 6.6.2005)

netindex	ipfrom	ipto	netname	hassubnets	gotanswer
1	3655335936	3656229167	.DTAG-DIAL15.AS3320.	0	Ripe
2	3640655872	3641016319	.DTAG-DIAL13.AS3320.	0	Ripe
3	1050279936	1050312703	.DTAG-BB16.AS3320.	0	Ripe
4	3154182144	3154247679	.WIN-IP.AS680.	0	Ripe
5	2213150720	2213216255	.UNIPADERBORN.AS680.AS1275.	0	Ripe
6	3645898752	3646494719	.DTAG-DIAL14.AS3320.	0	Ripe
7	1418199040	1421869055	.DTAG-DIAL20.AS3320.	0	Ripe
8	1347534848	1347535871	.DE-CIX-FRA-IXP.AS6695.	0	Ripe
9	3641996800	3641997055	.NETDIRECT-NET.AS28753.	0	Ripe
10	3645337344	3645337599	.LNC-DE-CUSTOMERLINKS2.AS13237.	0	Ripe
11	3645335808	3645336063	.EU-LAMBDANET-CORE-DE-P2P.AS13237.	0	Ripe
12	3645336576	3645336831	.EU-LAMBDANET-CORE-DE-CUS.AS13237.	0	Ripe
13	1050444800	1050447103	.DTAG-BB11.AS3320.	0	Ripe
14	3641996544	3641996799	.NETDIRECT-NET.AS28753.	0	Ripe
15	1381148928	1381236735	.ARCOR-DSL-NET12.AS3209.	0	Ripe
16	2449277952	2449278207	.ARCOR-BACKBONE-KOPPELNETZE-NET1.AS3211.	0	Ripe
17	2449345536	2449345791	.ARCOR-BACKBONE-ESN-NET1.AS3209.	0	Ripe
18	2449346560	2449346815	.ARCOR-BACKBONE-BB-NET1.AS3209.	0	Ripe
19	2262695936	2262761471	.EUNET-BACKBONE.AS286.	0	Ripe
20	2449346816	2449347071	.ARCOR-BACKBONE-BB-NET2.AS3209.	0	Ripe
21	2449347072	2449347327	.ARCOR-BACKBONE-BB-NET3.AS3209.	0	Ripe
22	2449276928	2449407999	.DE-ARCOR-20000314.AS3211.	0	Ripe
23	2172911616	2172977151	.ETH-ETHER.AS559.	0	Ripe
24	1091764224	1091780607	.CYBERMESA-COM.	0	Arin
25	3481927680	3481993215	.GBLX-8.	0	Arin
26	1125122048	1125253119	.GBLX-13.	0	Arin
27	3254490684	3254490879	.OPENTRANSIT-BACKBONE.AS5511.	0	Ripe
28	3254513664	3254517563	.OPENTRANSIT-BACKBONE.AS5511.	0	Ripe
29	1372691456	1372691711	.LNC-US-BB.AS13237.	0	Ripe
30	1388677120	1388677375	.LNC-UK-BB.AS13237.	0	Ripe
31	1388679168	1388679423	.LNC-FR-BB.AS13237.	0	Ripe
32	3492986880	3493068799	.GBLX-6D.	0	Arin
33	3492806656	3492823039	.GBLX-6A.	0	Arin
34	3589816320	3589827071	.TELIA-TIC-NET-1.AS1299.	0	Ripe
35	1119453184	1119461375	.ATDN.	0	Arin
36	3492864000	3492970495	.GBLX-6C.	0	Arin
37	3223411712	3223411967	.ETH-NET6.AS559.	0	Ripe
38	2184904704	2184970239	.SWITCH-LAN.AS559.	0	Ripe
39	3257544704	3257544959	.CH-TIX.AS8235.	0	Ripe
40	1347855104	1347855359	.LNC-CH-BB.AS13237.	0	Ripe
41	1042833408	1042837503	.GEANT.AS20965.	0	Ripe
42	2195783680	2195849215	.FSKNET-130-225.AS1835.	0	Ripe
43	3223715840	3223748607	.FSKNET-192-38.AS1835.	0	Ripe
44	1372695808	1372696063	.LNC-DK-BB.AS13237.	0	Ripe
45	3238675456	3238675711	.NORDUNET-BB-2.AS2603.	0	Ripe
46	3238722560	3238722815	.NORDUNET-BB-1.AS2603.	0	Ripe
47	3589424384	3589424639	.STOCKHOLM-CUSTOMER-LINKS.AS9057.AS3356.	0	Ripe
48	3589424128	3589424383	.STOCKHOLM-INFRASTRUCTURE.AS9057.AS3356.	0	Ripe
49	3569057792	3569057919	.EUROPE-MPLS-CORE.AS9057.AS3356.	0	Ripe
50	3279587328	3279587583	.FRANKFURT1-INFRASTRUCTURE2.AS9057.	0	Ripe

51	1044578304	1044643839	.DE-MIKNET-20001108.AS9057.AS3356.	0	Ripe
52	3279588352	3279588607	.HAMBURG-INFRA1.AS9057.	0	Ripe
53	67108864	83886079	.LVLT-ORG-4-8.	0	Arin
54	3261167616	3261169663	.STRATO-POOL-NET.AS5430.	0	Ripe
55	1401405440	1401421823	.VT-DYNAMICPOOL.AS12313.AS8881.	0	Ripe
56	3522428928	3522691071	.LEVEL3-CIDR.	0	Arin
57	3239489280	3239489535	.PL-AXIT.AS31310.	0	Ripe
58	1083703296	1084227583	.LC-ORG-ARIN.	0	Arin
59	1094189056	1094451199	.LC-ORG-ARIN-BLK2.	0	Arin
60	3567398656	3567398911	.TRANSFER-NETWORK1.AS9057.	0	Ripe
61	3646840832	3646947327	.DTAG-DIAL18.AS3320.	0	Ripe
62	1381113856	1381148927	.ARCOR-DSL-NET2.AS3209.	0	Ripe
63	1398759424	1398767615	.TDC-TELEDANMARK-BREDBAANDSADSL-NET.AS3292.	0	Ripe
64	1353089024	1353105407	.TDC-TELEDANMARK-BREDBAANDSADSL-NET.AS3292.	0	Ripe
65	1398276096	1398800383	.DK-TELEDANMARK-20031201.AS3292.	0	Ripe
66	3287875584	3287879423	.TELEDANMARK-IPNET.AS3292.	0	Ripe
67	1346326528	1346327551	.TELEDANMARK-IPNET.AS3292.	0	Ripe
68	3541827584	3542089727	.OPTUSINTERNET-AU.	0	APNIC
69	3398107136	3398115327	.OPTUSCOM-AU.	0	APNIC
70	3419439104	3419447295	.SINGTEL-IX.	0	APNIC
71	3419447296	3419455487	.SINGTEL-IX.	0	APNIC
72	1029177344	1029242879	.OPTUSCOM-AU.	0	APNIC
73	3275940352	3275940863	.LINX-PEER-2.AS5459.	0	Ripe
74	3287875584	3287941119	.DK-TELEDANMARK-980107.AS3292.	0	Ripe
75	3583285256	3583285256	.QSC-CUSTOMER-549812-195585.AS20676.	0	Ripe
76	3656253440	3656328447	.DTAG-DIAL17.AS3320.	0	Ripe
77	3289792512	3289809407	.AFRINIC-196-22-64-0.	0	Arin
78	3290296320	3290431487	.AFRINIC-196-29-240-0.	0	Arin
79	2461794304	2461859839	.UUNET-PIPEX.AS702.	0	Ripe
80	2650800128	2667577343	.EU-ZZ-158.AS1849.AS702.	0	Ripe
81	1411940352	1411940607	.PL-DIALOG.AS15857.	0	Ripe
82	3583279104	3583279359	.QSC-CORE.AS20676.	0	Ripe
83	3583279616	3583279871	.QSC-internal-Dus.AS20676.	0	Ripe
84	3583281664	3583282175	.QSC-IGP-4.AS20676.	0	Ripe
85	1394278400	1394327551	.NEOSTRADA-ADSL.AS5617.	0	Ripe
86	3575185920	3575186175	.POLPAK.AS5617.	0	Ripe
87	3575186688	3575186943	.POLPAK.AS5617.	0	Ripe
88	3268194048	3268194303	.TPNET-DO-OLSZTYN.AS5617.	0	Ripe
89	3281381376	3281383423	.TP-TRANSIT.AS29535.	0	Ripe
90	3645335552	3645335807	.LNC-DE-CUSTOMERLINKS2.AS13237.	0	Ripe
91	3254486016	3254486271	.OPENTRANSIT-BACKBONE.AS5511.	0	Ripe
92	3254491648	3254491903	.OPENTRANSIT-BACKBONE.AS5511.	0	Ripe
93	3254484992	3254485247	.OPENTRANSIT-BACKBONE.AS5511.	0	Ripe
94	3565723648	3565731839	.UUNET-BACKBONE.AS702.	0	Ripe
95	3279263104	3279263167	.ELBA.AS5617.	0	Ripe
96	3268192256	3268192511	.MAN-WROCLAW-1.AS5617.	0	Ripe
97	3284992000	3284992255	.TPSA-SZK.AS5617.	0	Ripe
98	3645336320	3645336575	.LNC-DE-CUSTOMERLINKS3.AS13237.	0	Ripe
99	1408434176	1408436223	.RRBONENET-1.AS28753.	0	Ripe
100	1394475008	1394540543	.NEOSTRADA-ADSL.AS5617.	0	Ripe
101	1087635456	1087897599	.GBLX-11D.	0	Arin
102	3652714496	3653238783	.DE-MEDIAWAYS-20010314.AS6805.	0	Ripe
103	3276214016	3276214271	.MWAYS-LOOPBACK.AS6805.	0	Ripe

104	3276234752	3276235263	.MWAYS-CORE.AS6805.	0	Ripe
105	3276267520	3276274431	.MWAYS-CORE.AS6805.	0	Ripe
106	3276275200	3276275455	.MWAYS-BACKBONE.AS6805.	0	Ripe
107	3574857728	3574923263	.DE-MEDIAWAYS-990820.AS6805.	0	Ripe
108	1350565888	1351786495	.DTAG-DIAL16.AS3320.	0	Ripe
109	1354301440	1354366975	.FREENET-POOL-NET.AS5430.	0	Ripe
110	1047051776	1047052031	.FREENET-POOL-NET.AS5430.	0	Ripe
111	1047052704	1047052735	.ROKA-NET.AS5430.	0	Ripe
112	1047052032	1047052287	.FREENET-POOL-NET.AS5430.	0	Ripe
113	2181038080	2197815295	.EU-ZZ-130.	0	Ripe
114	1404141568	1404149759	.FR-TELE2.AS1257.	0	Ripe
115	3566698496	3566709759	.EU-TELE2.AS1257.	0	Ripe
116	3276115968	3276116991	.AMS-IX.AS1200.	0	Ripe
117	3284066304	3284070399	.KPN-IO-IPC-NETWORK.AS286.	0	Ripe
118	1086586880	1086849023	.MCLEODUSA-00.	0	Arin
119	3470750464	3470750719	.EQUINIX-IX-ASH.	0	Arin
120	3522953216	3523215359	.MCLEODUSA-02.	0	Arin
121	3222339584	3222405119	.UNIV-IL.	0	Arin
122	3466985472	3467018239	.ICN6.	0	Arin
123	1356988416	1357053951	.DE-D2VODAFONE-20011212.AS15709.	0	Ripe
124	1410392064	1410392319	.NETDIRECT-NET.AS28753.	0	Ripe
125	1047055872	1047056127	.ROKA-NET.AS5430.	0	Ripe
126	3275939840	3275940351	.LINX-PEER-1.AS5459.	0	Ripe

## Anhang F Dateistruktur

```
PeerNear                                     // Hauptverzeichnis
|-- Client                                  // Die Client-Applikation
| |-- PNClient.py                          // Hauptdatei des Clients
| `-- config.py                            // Konfigurationsdatei des Clients
|-- Graph                                   // Graphrelevante-Dateien
| |-- gentraceroute.py                    // Programm zum Testen des Traceroute-Schedulers
| |-- graphapi.py                         // Adjazenz-Matrix-Klasse, Breitensuche, Dijkstra
| `-- represent.py                        // Eigentlicher Traceroute-Scheduler-Algorithmus
|-- JavaPeer                               // Der Java-Test-Peer
| |-- java.policy.applet                  // Sicherheitserlaubnis für Java
| |-- pnpeer                              // Quell-Dateien
| | |-- ITSocket.java                    // Wrapper aus Netgraph für Sockets
| | |-- PeerClientConnection.java        // Alle Funktionen für die Kommunikation zum Client
| | |-- PeerInetAddress.java            // Klasse aus NetGraph um mit IP-Adressen zu operieren
| | |-- TraceRoute.java                 // Traceroute ausführen und als Liste zurückliefern
| | |-- make.sh                          // Shell-Skript um Java-Applet zu kompilieren
| | |-- myKeystore                       // Datei mit Signierungsschlüssel
| | |-- pnpeerappl.java                 // Die Hauptdatei für die Kommandozeilenausführung
| | |-- pnpeermain.java                 // Die Hauptdatei für die Applet-Ausführung
| | `-- ssign.sh                        // Shellskript zum signieren des Applets
| |-- pnpeer.html                        // HTML-Template um Applet einzubinden
| `-- startappl.sh                       // Shell-Skript um Applet von der Kommandozeile zu starten
|-- Lib                                    // Bibliothek einiger Zusatzfunktionen
| |-- IPWrapper.py                       // Klasse um IP-Adressen in verschiedene Formate zu konvertieren
| |-- config.py                          // Einige Einstellungen welches Traceroute ausgeführt wird
| `-- future.py                          // Klasse um das ausführen von Threads leichter zu machen
|-- PeerNear.py                           // Wrapper um alle Klassen zusammenzuführen
|-- Server                                 // Server-Verzeichnis
| |-- PNServer.py                        // Die Hauptdatei des Server
| |-- config.py                          // Konfigurationsdatei für den Server
| |-- db                                  // Alle Datenbanken-Klassen
| | |-- config.py                       // Konfigurationsdatei mit alle Daten zur Datenbankanbindung
| | |-- connection.py                   // Instanz einer Verbindung zur MySQL-Datenbank
| | |-- networks.py                     // Funktionen für die Netzwerk-Tabelle
| | |-- peerclientdb.py                 // Funktionen für die Client-Tabelle
| | |-- peerdb.py                       // Funktionen für die Peer-Tabelle
| | |-- resetdb.py                     // Programm zum reinitialisieren der Datenbank
| | `-- whoisdb.py                      // Funktionen für die Whois-Tabelle
| |-- protocols.py                      // Client-<->Server-Protokoll Funktionen
| |-- whois.py                          // Klasse die Whois-Abfragen Koordiniert
| `-- whoisserver.py                   // Klassen um die verschiedenen Whois-Server zu bedienen
|-- Website                               // Verzeichnis der Webseite
| |-- PythonPoweredSmall.gif           // Kleines Python-Logo
| |-- appletdoc.htm                     // Anweisungen zum starten des Applets
| |-- cgi-bin
| | `-- makemap.cgi                    // CGI-Skript welches den Netzgraph neu generiert
| |-- cmdline.html                      // Anweisungen um das Test-Applet von der Kommandozeile zu starten
| |-- config.php                        // Datenbank-Konfiguration für die PHP-Skripte
| |-- dosome.php                        // PHP-Skript welches WHOIS-Daten von bestimmten Knoten anzeigt
| |-- favicon.ico                       // Icon von PeerNear
| |-- firefox_80x15.png                 // Firefox-Icon
| |-- index.html                        // Hauptdatei
| |-- intro.htm                         // Einleitende Worte
| |-- listpeers.php                     // PHP-Skript zum Anzeigen der momentan eingeklinkten Peers
| |-- listwhois.php                    // PHP-Skript zum Anzeigen der Liste aller bekannten Netze
| |-- news.htm                          // Neuigkeiten rund um PeerNear
| |-- papers.htm                        // Liste der wissenschaftliche Arbeiten
| |-- pearnear.gif                      // PeerNear-Logo
| |-- pnpeer.html                      // HTML-Seite um das Java-Applet zu starten
| |-- pnpeer.jar                        // Java-Test-Applet
| |-- site.css                          // Stylesheet für Seite
| |-- svginfo.htm                      // Informationen zum SVG-Format
|-- startall.sh                          // Shell-Skript: startet erst Server, dann Client
|-- startclient.sh                      // Shell-Skript: startet Client
`-- startserver.sh                      // Shell-Skript: startet Server
```

## Anhang G Konfigurationsdateien

### Anhang G.1 Server/config.py

```
ServerPwds={'test':'test'} # Passwort für die Clients zum connecten
ServerIP='217.20.118.113' # IP-Adresse des Servers
```

### Anhang G.2 Server/db/config.py

```
host="localhost" # Dieser Rechner
dbname="peernear" # Name der Datenbank
user="root" # Benutzername
passwd="" # Passwort
# Pfad um über das lokale Dateisystem Verbindung mit MySQL aufnehmen zu können:
unix_socket="/var/run/mysql/mysql.sock"
```

### Anhang G.3 Client/config.py

```
ServerIP='127.0.0.1' # Welche IP hat der Server, hier der gleiche Rechner
ServerPort=25000 # Port auf dem der Server Verbindungen entgegennimmt
ClientPort=25001 # Port auf dem der Client Verbindungen entgegennimmt
ServerUser='test' # Benutzername um an den Server connecten zu dürfen
ServerPass='test' # Passwort um an den Server connecten zu dürfen
PeerPwds={'testpeer':'testpwd'} # Password für die Peers um connecten zu dürfen
MaxConnections=700 # Maximale Anzahl an gleichzeitigen Verbindungen
MyIP='217.20.118.113' # IP-Adresse dieses Rechners
```

## **Ehrenwörtliche Erklärung**

Hiermit versichere ich, die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der von mir angegebenen Quellen angefertigt zu haben. Alle aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche gekennzeichnet. Die Arbeit wurde noch keiner Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Paderborn, 29. Juni 2005