



TECHNISCHE UNIVERSITÄT
CHEMNITZ

Fakultät für Informatik

Professur Rechnernetze
und verteilte Systeme

Diplomarbeit

Peer-To-Peer-Videostreaming

Autor: Daniel Schreiber

Betreuer: Prof. Dr. Uwe Hübner

Dr. Jörg Anders

Dr. Robert Baumgartl

Datum: 14. Juli 2005

Inhaltsverzeichnis

Titelseite	1
1 Aufgabenstellung	1
2 Einsatzzweck	2
2.1 Untersuchungsgegenstand	3
3 Streamingprotokolle	4
3.1 HTTP	4
3.2 RTP/RTCP	8
3.3 RTSP	9
4 Bestehende Peer-To-Peer Systeme	13
4.1 Peer-To-Peer-Netze mit zentraler Komponente	14
4.2 Reine Peer-To-Peer-Systeme	17
5 Konzept für ein Peer-To-Peer-System zur Übertragung von Streamingdaten	24
5.1 Architektur	24
5.2 Beschreibung des Trackerprotokolls	28
5.3 Unterstützung von DSL-Nutzern	30
5.4 Kostenabschätzung für den Tracker	31
6 Implementierung	36
6.1 Tracker	36
6.2 Proxy	36
6.3 Mplayer-glue	38
6.4 C-Tracker	39
7 Evaluierung der Streaminglösungen	41
7.1 Streaming über HTTP	41
7.2 Streaming über RTSP/RTP	43
7.3 Peer-To-Peer-Netz mit Trackerimplementierung in Perl	43
7.4 Peer-To-Peer-Netz mit Trackerimplementierung in C	54
8 Zusammenfassung	59

A	Tabellierte Messergebnisse	60
B	Befehlsreferenz	64
B.1	Tracker	64
B.2	Proxy	64
B.3	Mplayer-glue	65
C	Hardwareumgebung	66
C.1	Rechnerspezifikation	66
C.2	Netz	66
D	Glossar	68

1 Aufgabenstellung

Entwicklung eines Peer-To-Peer-Netzwerkes für Videostreaming - Netzwerkteil

Für Übertragung von Live-Videodaten an mehrere Empfänger gibt es als etablierte Technologien die Übertragung per Multicast an mehrere Empfänger sowie die Unicast-Übertragung an jeden Empfänger von einem Verteilpunkt aus. Nachteilig ist im ersten Fall, dass viele ISPs kein Multicast unterstützen, im zweiten Fall der hohe Bandbreitenbedarf am Verteilpunkt. Die Unterschiede in der verfügbaren Bandbreite von ISP-Zugängen (DSL, ISDN) und Campusnetzwerken (z.B. Studentennetzen) erfordern es, das Videomaterial in mehreren Qualitätsstufen anzubieten. Im Team soll ein System entworfen und realisiert werden, das aus einer Quelle effizient verschiedene Qualitätsstufen des Videomaterials erzeugt („Videoteil“) und dieses mittels Peer-To-Peer-Technologie verteilt („Netzwerkteil“).

Aufgabenstellung „Netzwerkteil“

- Untersuchung der Faktoren, die die Skalierbarkeit von Unicast-Lösungen beschränken, Recherche existierender Skalierbarkeits-Verbesserungen
- Untersuchung bestehender Peer-To-Peer-Technologien unter dem Aspekt der Eignung für Videostreaming
- Entwicklung bzw. Ableitung einer Peer-To-Peer-Lösung, die Echtzeitvideoströme übertragen kann und Caching-Funktionen ausführen kann
- Untersuchung des Latenzzeit-Verhaltens und anderer relevanter Parameter

Die entstandenen Programme sind unter eine Open-Source-Lizenz zu stellen.

2 Einsatzzweck

Beim Verteilen von Videodaten gibt es zwei verschiedene Einsatzszenarien: Live- und On-Demand-Übertragung. Bei der Liveübertragung werden die Videodaten mit möglichst geringer Verzögerung zum Empfänger übertragen. Alle Empfänger einer Liveübertragung bekommen nahezu gleichzeitig dieselben Daten. Bei einer On-Demand-Übertragung fordern die Teilnehmer verschiedene Teile eines aufgezeichneten Medienstroms an. Das heißt, dass zwei Teilnehmer der Übertragung nicht zwangsläufig zur gleichen Zeit denselben Inhalt zu sehen bekommen. Bei einer On-Demand-Übertragung besteht auch die Möglichkeit, im Medienstrom zu springen („vor- und zurückspulen“).

Der Hintergrund der Aufgabenstellung ist die Verteilung von Vorlesungen. Dabei ist Liveübertragung nützlich, um Interaktion zwischen Lesendem und Zuschauern möglich zu machen, beispielsweise über eine Konferenz auf einem Jabber- oder IRC-Server. Denkbar wäre aber auch eine Telefonkonferenz über Voice-over-IP oder ein herkömmliches Telefonnetz. Neben Liveübertragungen ist auch On-Demand-Zugriff sinnvoll, zum Beispiel um zur Prüfungsvorbereitung einzelne Vorlesungen noch einmal anzusehen. Dazu müssen aufgenommene Vorlesungen gespeichert werden. Um Interaktivität zu ermöglichen, soll die Zeit zwischen Aufnahme und Wiedergabe (Latenzzeit) bei Liveübertragungen so gering wie möglich sein.

Bei Vorlesungen ist es nützlich, mehrere Videoquellen verwalten zu können, beispielsweise um mit einer Kamera den Vorlesenden abzubilden, mit einer anderen Kamera Tafelbild oder Präsentationsfolien zu übertragen und bei Diskussionen das Publikum im Gespräch mit dem Vortragenden zu zeigen.

Neben Vorlesungen sind auch andere Anwendungen möglich, zum Beispiel die Übertragung von Studentenfernsehen, was sich damit preiswerter als über Funk oder Übertragung in Kabelnetzen realisieren lässt.

Die verfügbare Bandbreite im Internet variiert sehr stark. Universitäten sind in der Regel mit hoher Bandbreite in Forschungsnetzen mit dem Internet verbunden. An die Netze der Universitäten sind oft auch Studentennetze angeschlossen, die Studentenwohnheime mit breitbandigem Internetzugang (zur Zeit meist 10 oder 100 MBit/s Ethernet) versorgen. Die breite Masse der Bevölkerung verfügt dagegen üblicherweise über nicht mehr als einen DSL-Anschluss oder sogar nur über einen für Videoübertragung ungeeigneten ISDN-Zugang. Das System soll sowohl für Anwender in Studentenwohnheimen als auch für Nutzer mit DSL-Anschlüssen geeignet sein. Weiterhin soll das System zumindest auf Endanwenderseite leicht auf verschiedene Plattformen portierbar sein.

2.1 Untersuchungsgegenstand

Aus dem Einsatzzweck leiten sich verschiedene Aspekte ab, die untersucht werden sollen. Videoübertragung und -wiedergabe sind zeitkritische Anwendungen, die weiche Echtzeitanforderungen stellen. Das heißt, dass bei Verletzung des Echtzeitkriteriums eine Qualitätsverschlechterung des angebotenen Dienstes, aber kein kritischer Fehler, vorliegt.

Insbesondere die Skalierbarkeit des Systems, die Zeit zwischen Anforderung und Abspielen des Videos und das Verhalten des Systems im Fehlerfall (Ausfall einzelner Komponenten) sind untersuchenswert. Die Skalierbarkeit eines Systems gibt an, wie viele Teilnehmer ein System mit welchen Kosten maximal versorgen kann und ist insbesondere für den Anbieter interessant. Die Zeit zwischen Anforderung und Abspielen ist interessant, weil der Nutzer von Fernsehen und Homevideo (VHS, DVD, etc.) gewohnt ist, dass sofort nach Anforderung das Video abgespielt wird. Weiterhin ist die Veränderung dieses Parameters bei besonderen, außergewöhnlichen Ereignissen (Beispiel: 11. September 2001) interessant, wenn ungewöhnlich viele Nutzer zur gleichen Zeit mit der Dienstnutzung beginnen wollen. In dem Fall soll das System den Dienst bis zur physikalischen Leistungsgrenze anbieten und nicht aufgrund der vielen gleichzeitigen Anfragen weniger Teilnehmer versorgen können. Bei der Untersuchung des Verhaltens im Fehlerfall ist insbesondere interessant, wie das System auf zufällige Verschlechterung der Übertragungsgüte (Paketverluste) reagiert.

3 Streamingprotokolle

Dieses Kapitel soll verbreitete offene Streamingsysteme vorstellen und unter den in Kapitel 2.1 beschriebenen Aspekten untersuchen.

3.1 HTTP

HTTP [1] ist ein Protokoll, das zur Dateiübertragung zwischen zwei Rechnern („Client“ und „Server“) entworfen wurde. Es kann Dateien in Teilabschnitten übertragen und dabei Proxies und Caches zur Leistungssteigerung benutzen. Es gibt stabile Implementierungen von Clients, Proxies und Servern für nahezu jede Systemplattform. HTTP benötigt ein zuverlässiges Transportprotokoll, fast immer wird TCP [2] verwendet. Durch die Voraussetzung eines zuverlässigen Transportprotokolls wird die Implementierung eines cachenden Proxies erleichtert.

Die Kommunikation zwischen Client und Server besteht bei HTTP aus zwei Phasen: Einer Anfragenachricht (Request) und der darauf folgenden Antwortnachricht (Response) vom Server. Nachdem die Antwort empfangen wurde, kann der Client weitere Anfragen senden. Das Ende der Kommunikation wird durch das Schließen der Transportverbindung signalisiert. Jede Nachricht besteht aus 3 Teilen: einer Zeile, in der die Anfrage mit URL angegeben ist bzw. der Antwortcode mit Status bei einer Antwort, mehreren Kopfzeilen (Header) und – durch eine Leerzeile getrennt – dem Inhalt. Bei der Anfrage wird anhand des übertragenen URL die betroffene Ressource identifiziert. In den Headerzeilen können weitere Attribute übergeben werden, die kennzeichnen, auf welche Art und Weise mit der Ressource umgegangen werden soll.

Obwohl HTTP nur für Dateitransfer entworfen wurde, wird es auch zur Übertragung von Medienströmen genutzt. Dabei wird bei der Liveübertragung als Antwort auf einen Request der aktuell abzuspieldende Inhalt geliefert. Dieser wird solange gesendet, bis der Client die Übertragung abbricht. Beim On-Demand-Streaming über HTTP wird die Mediendatei von Beginn an übertragen – wie bei einem normalen Download. Obwohl Daten in Teilstücken übertragen werden können, kann in einem HTTP-Stream normalerweise nicht gesprungen werden, da HTTP lediglich Bytes adressieren kann. Wenn der Server eine entsprechende Unterstützung zur passenden Aufbereitung der Daten anbietet, sind auch Sprünge möglich. Dies könnte zum Beispiel über Kodierung der abzurufenden Teile im URL oder über herstellerspezifische Einträge im Kopf geschehen. Bei jedem Sprung muss ein neuer Request an den Server geschickt werden, damit dieser dann die passenden Daten liefern kann. Da es keine Standards für die Kodierung der Adressierung von Videodaten in einer Ressource außer über Bytes gibt, ist eine solche Lösung kaum interoperabel. Nach dem Ende eines On-Demand-Downloads kann der Medienstrom wie

eine lokale Datei verwendet werden. Das bedeutet, dass das Medienformat Unterstützung für Sprünge anbieten muss, damit der Client andere Positionen anspringen kann.

3.1.1 Skalierungsverbesserungen für HTTP

Um die Anzahl der maximal gleichzeitig versorgbaren Clients zu erhöhen, gibt es verschiedene Möglichkeiten wie zum Beispiel DNS Round Robin, Proxy Caches und Loadbalancer. Diese sollen nun mit Berücksichtigung der Tauglichkeit für Medienübertragung vorgestellt werden.

Beim *DNS Round Robin* werden einem Hostnamen mehrere A-Records zugewiesen, d.h. der Name wird in verschiedene IP-Adressen aufgelöst. Wenn die Gültigkeitsdauer der DNS Records hoch ist, dauert es lange, bis neu hinzugefügte Server benutzt werden können. Damit kann nur mit Verzögerung auf Spitzenlastsituationen reagiert werden. Außerdem gibt es keine festgelegten Algorithmen zur Verteilung der Anfragen, die die Belastung der einzelnen Server beachten. Jeder Client entscheidet selbst, welche IP-Adresse er als Ziel verwendet. Durch die Verwendung verschiedener IP-Adressen ist eine globale Verteilung der Server möglich.

Loadbalancer können mehrere Server hinter einer IP-Adresse verstecken. Dazu gibt es verschiedene Möglichkeiten, die einfachste ist Network Address Translation (NAT). Dabei ist der Loadbalancer NAT-Gateway mit der IP-Adresse, unter der der Dienst angeboten wird. Die Server befinden sich hinter dem Gateway in einem privaten Subnetz. Das Gateway übersetzt die Ziel-IP-Adresse und den Port auf IP-Adresse/Port eines der Server. Dieser schickt die Antwort über das Gateway zurück an den Client. Vorteil dieser Methode ist die Einfachheit. Nachteilig ist, dass die gesamte Bandbreite aller Server vom Gateway verarbeitet werden muss. Bei bandbreitenintensiven Anwendungen wie Videostreaming, bei denen der Flaschenhals nicht die Verarbeitungsleistung der Server, sondern die Bandbreite ist, stellt dies daher keine gute Wahl dar.

Eine weitere Möglichkeit zum Loadbalancing besteht im Einrichten von Tunneln. Dabei wird zwischen dem Loadbalancer und den Servern ein IP-Tunnel aufgebaut. Die Server haben auf ihren Tunnel-Interfaces die IP-Adresse konfiguriert, unter der der Dienst erreichbar ist. Der Loadbalancer hat ebenfalls diese IP-Adresse an einem vom Internet erreichbaren Interface eingestellt. Pakete, die an die Dienst-IP-Adresse gerichtet sind, werden an den Loadbalancer geroutet. Dieser verschickt diese Pakete durch einen der Tunnel an den zugehörigen Server weiter. Der Server sendet die Antwort unter Benutzung seiner Routingtabelle. Deren Einträge sollten so gestaltet sein, dass die Antwortpakete einen anderen Weg als durch den Tunnel nehmen. Dadurch entsteht beim Loadbalancer kein Engpass mehr, weil dieser nur die Anfragen verarbeiten muss. Bei Aufgaben, bei denen die Anfragen sehr kleine Datenraten benötigen, die Antworten jedoch hohe, ist diese Konstellation sinnvoll. Ein großer Vorteil der Verwendung von Tunneln ist die Möglichkeit zur geografischen Verteilung der Server. Allerdings dürfen dann die Router auf dem Weg zwischen Server und Client die Absender-IP-Adresse mit ihrer Routingtabelle nicht vergleichen (reverse path filter).

Eine weitere Möglichkeit zum Loadbalancing ist MAC Address Translation (MAT). Dabei haben sowohl Server als auch Loadbalancer die gleiche IP-Adresse und befinden sich im gleichen Subnetz. Die Server müssen sich passiv verhalten und dürfen nicht auf ARP-Anfragen antworten, weil dies zu Adresskon-

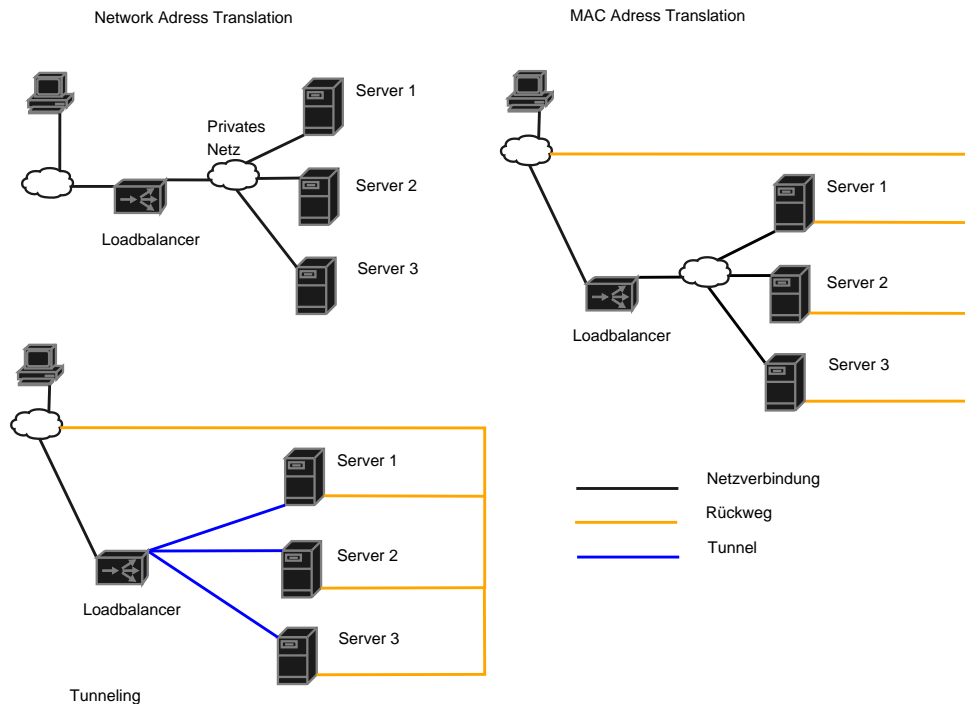


Abbildung 3.1: Verschiedene Möglichkeiten des Loadbalancings

fiktionen führen würde. Jedes Paket, das an die IP des Dienstes geschickt wird, erreicht den Loadbalancer. Dieser ändert die Ziel-MAC-Adresse des Paketes und schickt es erneut ins Subnetz. Die IP-Adresse bleibt erhalten. Das heißt, oberhalb von der Subnettschicht werden keine Daten im Paket geändert.

Plötzlichen Überlastsituationen kann durch Hinzufügen von Servern in die Liste der zur Verfügung stehenden Server begegnet werden. Die maximale Anzahl an Teilnehmern wird allerdings durch die Leistung des Loadbalancers begrenzt, abhängig von der Art und Weise des Loadbalancings durch Netzbandbreite (bei NAT) oder durch Verarbeitungsleistung (bei Tunnel oder MAT). Die Netzbandbreite wird insbesondere dann zum Problem, wenn viele Teilnehmer gleichzeitig versorgt werden müssen. Dagegen stellt die Verarbeitungsleistung ein Problem beim Flash-Crowd-Syndrom dar, da dann innerhalb sehr kurzer Zeit viele Anfragen durch den Loadbalancer verteilt werden müssen.

Caches sind bei der Verringerung der Serverlast bei Live-Inhalten nicht hilfreich, weil der Inhalt der Daten vom Zeitpunkt des Abrufs abhängt und damit nicht cachebar ist. Die zentrale Eigenschaft von Caches ist aber, dass Inhalte, die schon früher abgerufen wurden, unverändert wiedergegeben werden. Damit würde ein Nutzer bei einer Liveübertragung bei Abbruch und anschließender Neuansforderung des Streams die gleichen Daten geliefert bekommen, die er zuvor schon empfangen hatte. Caches könnten maximal den einmal vom Server empfangenen Inhalt an mehrere Teilnehmer gleichzeitig verteilen, dies entspräche dann aber nicht mehr der in [1] definierten Semantik.

Eine weitere Möglichkeit ist die Verwendung von *Redirects*. Dabei wird eine Adresse allen Clients bekannt gegeben. Diese erhalten als Antwort eine andere Adresse, unter der die geforderte Ressource ver-

	DNS RR	Loadbalancing			Caches	Redirect
		NAT	Tunnel	MAT		
Geeignet für Streaming	gut	mäßig	gut	gut	schlecht	gut
Reaktion auf Spitzenlast	schlecht	gut	gut	gut	schlecht	gut
geografische Verteilung	ja	nein	ja	nein	ja	ja
Flaschenhals	Reaktionszeit	Netzbandbreite	Verarbeitungslistung	Verarbeitungslistung	Verarbeitungslistung	Verarbeitungslistung

Tabelle 3.1: Skalierungsverbesserungen für HTTP

fügar ist. Man kann nun dieselbe Ressource unter verschiedenen Adressen (auf verschiedenen Servern) bereitstellen. Dabei werden die zugewiesenen Adressen passend ausgeteilt. Es kann verschiedene Schedulingstrategien dafür geben. Um beim Flash-Crowd-Syndrom nicht zum Flaschenhals zu werden, sollten die Strategien aber möglichst wenig Rechenzeit benötigen.

Verhalten bei schwankenden Übertragungsraten HTTP erbt bei schwankenden Übertragungsraten und Paketverlusten die Eigenschaften des Übertragungsprotokolls. Im Falle von TCP bedeutet das, dass bei Paketverlust auf die erneute Übertragung des verloren gegangenen Paketes gewartet werden muss. Durch die Sliding-Window-Technik werden die unmittelbaren Auswirkungen eines einzelnen Paketverlustes gemindert. Bei wiederholtem Paketverlust reduziert TCP die Übertragungsrate. Sofern diese dann unter die Datenrate des Medienstroms sinkt, stockt die Medienwiedergabe. Dieser Effekt kann durch den Einsatz von Puffern in der Wiedergabesoftware gemindert werden. Falls der Puffer wegen der zu niedrigen Übertragungsrate entleert wurde, kommt es natürlich trotzdem zu Aussetzern bei der Wiedergabe.

TCP ist ein synchrones Übertragungsprotokoll. Das bedeutet, dass die beiden Übertragungspartner solange keine Daten mehr senden können, wie alle Pakete im Sliding Window noch bestätigt werden müssen. Der sendende Server bemerkt dies – sein Sendebefehl würde blockieren oder einen Fehler liefern. Daraufhin könnte er die Datenrate des Medienstroms reduzieren. Dies läuft allerdings dem Caching zuwider, da der übertragene Medienstrom dann abhängig von der jeweiligen Verbindung wäre. Wenn der Server eine entsprechende (netz-)lastabhängige Aufbereitung der Daten vornehmen würde, wäre dessen Belastung in einem solchen Szenario sehr hoch. Man könnte dann bei ausreichender Prozessorleistung und Knappheit an Netzbandbreite mehr Empfänger versorgen, da dann Paketverluste auftreten würden und TCP in der Folge automatisch die Datenrate reduziert. Dies stellt also eine Qualitätsverschlechterung beim Empfänger zugunsten einer höheren Teilnehmeranzahl dar.

0 0 0 1	0 2	0 3	0 0 0 0 4 5 6 7	0 8	0 1 1 1 1 1 9 0 1 2 3 4 5	1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1	
V	2	P	X	CC	M	PT	sequence number
timestamp							
synchronization source (SSRC) identifier							
contributing source (CSRC) identifiers							

Abbildung 3.2: RTP-Header

3.2 RTP/RTCP

Um die mit TCP verbundenen Probleme zu vermeiden, wurde mit dem Realtime Transport Protocol (RTP) eine Möglichkeit entwickelt, Streamingdaten im Internet zu übertragen. RTP realisiert eine Schicht oberhalb von UDP und stellt folgende Funktionen zur Verfügung:

- Zeitliche Ordnung der Daten (Zeitstempel)
- Logische Ordnung von RTP-Paketen (Sequenznummern)
- Multiplexing mehrerer Medienströme
- Unterscheidung der Nutzdaten nach verschiedenen Typen

In Abbildung 3.2 ist der RTP-Header zu sehen. In [3] ist die genaue Bedeutung der einzelnen Felder spezifiziert.

Auf den ersten Blick mag die Existenz von Zeitstempeln und Sequenznummern redundant erscheinen, jedoch haben beide eine unterschiedliche Semantik. Die Sequenznummern zählen die RTP-Pakete durch. Der Zeitstempel kennzeichnet den Zeitpunkt zu dem ein Sample der zu übertragenden Daten aufgenommen wurde. Falls ein Sample jeweils in ein RTP-Paket passt, ändern sich Sequenznummer und Zeitstempel in jedem Paket. Sollte das Sample allerdings größer als ein RTP-Paket sein, haben mehrere Pakete den gleichen Zeitstempel, aber verschiedene Sequenznummern. Das Ende von zusammen gehörenden RTP-Paketen wird durch ein gesetztes Marker-Bit (M) im letzten Paket signalisiert.

Da das Internet Pakete nach dem Best-Effort-Prinzip vermittelt, können keine Garantien über Zustellungszeiten gemacht werden. Der Empfänger eines RTP-Datenstroms muss daher puffern, um Schwankungen der Zustellungszeiten auszugleichen. In Abbildung 3.3 ist dies dargestellt. RTP 2 kommt zu früh und wird einfach an den Puffer angehängt. RTP 3 kommt erst nach seinem Abspielzeitpunkt beim Empfänger an. Daher kann dieses Paket nicht abgespielt werden und wird verworfen.

Mit dem Resource Reservation Protocol (RSVP) [4] kann Bandbreite auf dem Übertragungsweg reserviert werden. Zusammen mit Quality-of-Service-Einstellungen auf den Routern kann so eine vereinbarte Übertragungsqualität gewährleistet werden. Allerdings ist die praktische Verbreitung im Internet in der

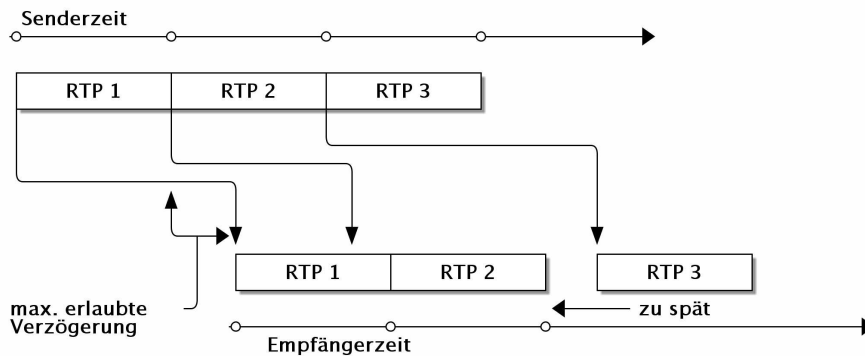


Abbildung 3.3: Zeitliche Ordnung von RTP-Paketten

Regel auf Campusnetze begrenzt, wenn es überhaupt eingesetzt wird. Netzübergreifender Einsatz von RSVP und QoS ist im Internet die Ausnahme und hat einen ähnlich geringen Verbreitungsgrad wie Multicast. Dafür gibt es mehrere Gründe: Erstens ist die Implementierung teuer, weil in den Routern Zustandsinformationen gespeichert werden müssen. Dies verbraucht teuren Routerspeicher. Das normale (Unicast-)Routing im Internet ist zustandslos und daher preiswert zu implementieren. Zweitens ist die Konfiguration der Router kompliziert – falsche Konfiguration kann sogar zu Qualitätsverschlechterungen führen. Drittens gibt es keine anbieterübergreifenden Abrechnungsstandards für verschiedene Dienstqualitäten. Das Ziel, eine definierte Übertragungsqualität zu gewährleisten, kann aber nur erreicht werden, wenn die Qualität entlang des gesamten Weges garantiert werden kann. Und schließlich kann viertens die Qualität nur garantiert werden, solange die Routingwege stabil sind. RSVP reserviert Bandbreiten nur entlang eines Weges. Sobald sich der Weg ändert, kennen die neu hinzugekommenen Router die Qualitätsanforderungen noch nicht, bis eine erneute Reservierung erfolgt. Aus diesem Grund verlangt RSVP, dass Reservierungen periodisch aufgefrischt werden müssen.

RTP-Datenströme sind unidirektional. Das heißt in diesem Fall, dass der Sender die Daten unabhängig davon sendet, ob der Empfänger sie überhaupt empfangen kann. Um trotzdem eventuell eine Anpassung der Datenrate zu erreichen, gibt es RTCP, das zusammen mit RTP in RFC 1889 [3] definiert wurde. Mittels RTCP sendet der Empfänger unter anderem Statistiken über die empfangene Qualität an den Sender. Dieser kann dann in einer geeigneten Weise darauf reagieren, indem er zum Beispiel die Bitrate eines übertragenen Videos reduziert. Die Unidirektionalität von RTP erlaubt es, neben Unicast auch Multicast zur Übertragung zu benutzen. Das bedeutet, dass mehrere Empfänger denselben Medienstrom empfangen, er aber nur einmal vom Sender abgeschickt wird. Die Replikation der Daten führen die Multicasterouter im Netz durch.

3.3 RTSP

Das Realtime Streaming Protocol (RTSP [5]) dient zur Steuerung von RTP-Datenströmen. Verglichen mit einem klassischen Videorecorder übernimmt es die Rolle der Fernbedienung. RTSP wird oft über

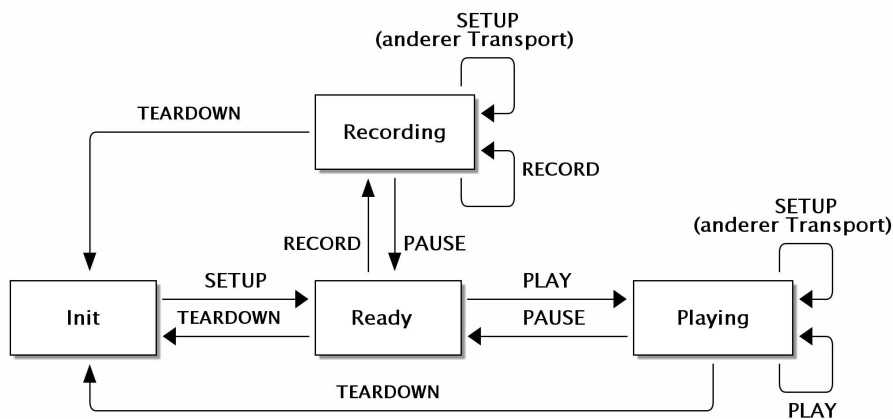


Abbildung 3.4: Zustandsübergänge RTSP

eine TCP-Verbindung zwischen Client und Server realisiert. Aber auch UDP [6] oder RDP [7] sind prinzipiell möglich und erlaubt. RTSP ist ein zustandsbasiertes Protokoll mit folgenden Zuständen und den in Abbildung 3.4 dargestellten Übergängen:

Zustand	Client	Server
Init	SETUP wurde gesendet. Warte auf Antwort. Anfangszustand.	Kein SETUP wurde empfangen. Anfangszustand.
Ready	Antwort auf SETUP wurde empfangen oder Antwort auf PAUSE wurde im Playing Zustand empfangen.	SETUP oder PAUSE wurde beantwortet.
Playing	Antwort auf PLAY wurde empfangen.	Antwort auf PLAY wurde gesendet. Daten werden übertragen.
Recording	Antwort auf RECORD wurde empfangen.	Der Server zeichnet Daten auf.

Da RTSP kein zuverlässiges Transportprotokoll wie TCP fordert und trotzdem zustandsbasiert arbeitet, müssen die einzelnen Pakete einer Verbindung zugeordnet werden können. Diese Zustandszuordnung wird über einen Sitzungsidentifikator realisiert. Der Zustand einer RTSP-Verbindung zwischen Client und Server wird also nicht über das Bestehen einer TCP-Verbindung signalisiert. Es ist durchaus möglich und zulässig, dass für jeden Request eine neue TCP-Verbindung aufgebaut wird. Es ist ebenfalls zulässig, dass Sitzungsidentifikatoren an einen anderen Client weitergegeben werden. Dies ist bei Multicastanwendungen nützlich. Es kann also wie beim klassischen Videorecorder die Fernbedienung an einen anderen Zuschauer weitergereicht werden. Die Aushandlung der Zugriffsrechte und das Weiterleiten des Sitzungsidentifikators ist nicht Bestandteil von RTSP und muss mit einem anderen geeigneten – möglicherweise nichttechnischen – Protokoll realisiert werden.

Das Nachrichtenformat von RTSP ähnelt dem von HTTP [1]. Anfrage- und Antwortnachrichten bestehen jeweils aus 3 Teilen: Befehl bzw. Antwort(-code), Kopf (Header) und einem möglicherweise leeren Dokument. Es findet eine Anfrage-Antwort-Paarung statt, wobei anders als bei HTTP mehrere Anfragen hintereinander geschickt werden können, ohne dass zuvor eine Antwort abgewartet werden muss (Pipelining). Damit Anfragen und Antworten einander zugeordnet werden können, bekommt jedes Anfrage-Antwort-Paar eine Sequenznummer. Diese wird im `CSeq`-Feld im Kopf übertragen. Der Sitzungsidentifikator wird vom Server zugeteilt und im `Session`-Feld im Kopf übertragen. Sobald der Client vom Server einen Sitzungsidentifikator übermittelt bekommen hat, muss er diesen für jede zukünftige Kommunikation mit dem Server verwenden, bis die Sitzung beendet wurde.

Als Steuerungsprotokoll muss RTSP den Datenübertragungsweg aushandeln. Üblicherweise werden die Daten über RTP übertragen. Es ist allerdings auch möglich, die Nutzdaten direkt in RTSP einzubetten. Dies ist allerdings nur für den Fall der Verwendung von RTSP über TCP definiert und als Notlösung zur Überwindung von Firewalls und NAT gedacht. Mit Verwendung von TCP ergeben sich natürlich auch die negativen Eigenschaften bei schwankender Netzqualität, die schon im Abschnitt über HTTP besprochen wurden. Falls RTP vereinbart wurde, gibt es verschiedene Möglichkeiten der Aushandlung, um auch Multicast-Anwendungen zu unterstützen. Die Aushandlung geschieht über das `Transport`-Feld im Kopf. Dabei kann der Client Wünsche an den Server richten, die der Server erfüllen kann, aber nicht muss.

3.3.1 Skalierbarkeit

Es gibt verschiedene Ansätze, die Skalierbarkeit von RTSP/RTP-Streaminglösungen zu verbessern (Abb. 3.5). Der Standardfall ist die Verwendung eines einzigen Servers zur Steuerung (RTSP) und zur Datenauslieferung (RTP). Wenn die Netzbandbreite eines einzelnen Servers zur Datenauslieferung nicht mehr reicht, kann die Funktionalität der Steuerung und Datenauslieferung auf getrennte Server verteilt werden. Da zur Steuerung nicht viele Daten übertragen werden müssen, können mehrere RTP-Server von einem RTSP-Server gesteuert werden. Wenn lediglich Liveinhalte verteilt werden sollen, kann auch Multicast zur Verteilung verwendet werden, sofern das Netz Multicast anbietet. Dann hat jeder Client eine Verbindung zum RTSP-Server und empfängt die RTP-Daten über Multicast. Eine weitere Möglichkeit ist die Benutzung von Proxy-Caches.

Dabei ist zu beachten, dass die meisten RTSP-Operationen an sich nicht cacheable sind, da sie sich immer konkret auf eine Ressource und die Zustandsassoziation beziehen, die manipuliert wird. Allerdings können die übertragenen Medienströme gecached werden. Dabei arbeitet ein cachender Proxy wie ein RTSP-Server. Das heißt, er speichert Medienströme, die er vom ursprünglichen RTSP-Server bekommen hat, zwischen und liefert sie an seine Klienten aus. Hat er das angeforderte Medienintervall bereits vorrätig, kann er es aus seinem Speicher an den Klienten ausliefern. Dabei sind auch Kombinationen möglich, also teilweise Auslieferung aus dem Speicher und Auslieferung der fehlenden Teile nach Download vom ursprünglichen Server. Dabei müssen sehr wahrscheinlich Daten wie SSRC, Sitzungsidentifikator usw. angepasst werden. In [8] wird detailliert auf die Anforderungen eingegangen, die bei der Implementierung eines solchen Systems berücksichtigt werden müssen. In diesem Papier wurden auch Performancemessungen beschrieben, die belegen, dass entsprechende Proxy-Caches die Startup-

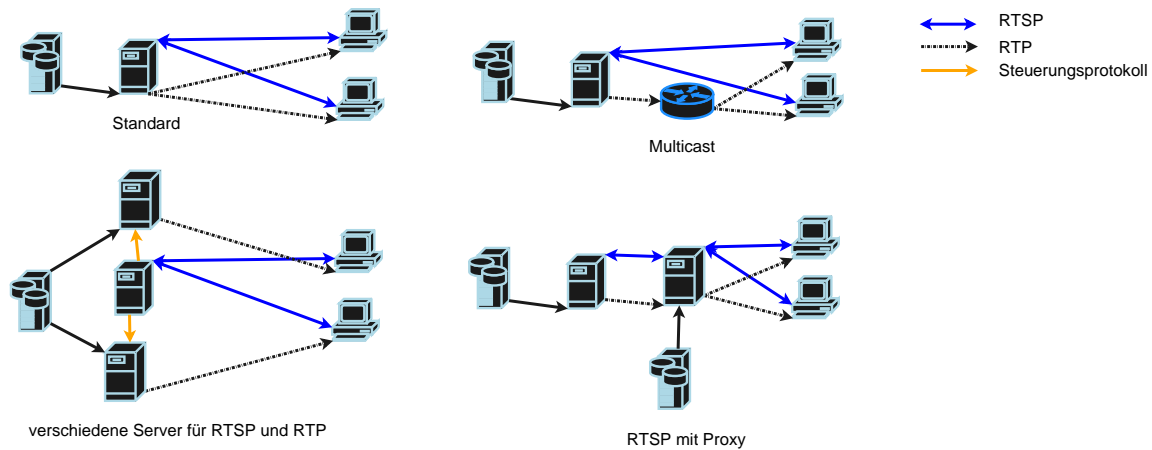


Abbildung 3.5: Verschiedene Möglichkeiten zur Verbesserung der Skalierbarkeit von RTSP

zeit für Streamingssessions reduzieren können, die Netzwerklast verteilen und somit die Skalierbarkeit erhöhen. Die Startupzeiten konnten insbesondere dadurch reduziert werden, dass die ersten Sekunden eines Videoclips bis zu 10-mal schneller an den Client übertragen wurden als es für das Abspielen nötig wäre. Gegen das Flash-Crowd-Syndrom hilft das jedoch nur eingeschränkt, da in diesem Fall besonders viele Teilnehmer anfangen, einen Clip abzuspielen und deshalb das schnellere Füllen der Puffer nicht möglich ist, weil das Netzwerk gesättigt sein kann. Durch den überhöhten Ansturm kann sich die Qualität der Übertragung verringern, weil Pakete verloren gehen und eventuell Messungen der Clients über die Netzqualität verfälscht werden. Wenn die Clients anhand der gemessenen Paketverluste die Größe ihrer Puffer berechnen, dann hat es sogar den gegenteiligen Effekt, falls die Clients die Puffer weiter vergrößern. Dieser Effekt könnte über eine Aushandlung der Puffergröße zwischen Client und Server oder Proxy verhindert werden. Dies ist aber im Protokoll nicht vorgesehen.

4 Bestehende Peer-To-Peer Systeme

In Kapitel 3 wurden bestehende Streamingtechnologien vorgestellt und wurde auf die Schwierigkeiten in Bezug auf die Anforderungen aus Kapitel 2 hingewiesen. In diesem Kapitel sollen bestehende Anwendungen der Peer-To-Peer-Technologie vorgestellt werden, da die Peer-To-Peer-Technologie skalierbare kostengünstige Verteilungsmechanismen verspricht. Im Rahmen der Vorstellung sollen folgende Punkte beleuchtet werden:

Architektur des Peer-To-Peer-Systems

- Wie werden Peers gefunden?
- Wie werden Peers ausgewählt?
- Wie werden (Nutz-)Daten gefunden?
- Wie wird dafür gesorgt, dass das System robust ist?
- Wird guter Durchsatz erzielt? Wenn ja, wie?

Routing

- Nach welchen Kriterien werden die Peers ausgewählt, mit denen Kontakt aufgenommen wird?
- Nach welchen Kriterien werden Nutzdaten weitergeleitet?

Eignung für Streaming

- Kann das Peer-To-Peer-System mit Daten unbekannter Länge umgehen?
- Wie kann die Übertragung des Datenstroms fortgesetzt werden, wenn ein Daten liefernder Peer ausfällt?

Der Peer-To-Peer-Begriff erlebt in letzter Zeit in der Wissenschaft und den Medien einen regelrechten Boom. Dabei werden oft sehr verschiedene Dinge als Peer-To-Peer bezeichnet, so dass der Begriff eher unscharf verwendet wird. Im Folgenden soll daher auf die Kriterien aus [9] zurückgegriffen werden:

1. Kann das untersuchte System mit zeitlich begrenzten Netzverbindungen und sich ändernden IP-Adressen der Teilnehmer umgehen?
2. Haben die Knoten am Rand des Netzes signifikante Autonomie?

Nur wenn beide Kriterien erfüllt sind, dann handelt es sich um eine Peer-To-Peer-Anwendung. Das bedeutet, dass der Hauptteil der Arbeit von der Anwendung am Rand des Netzes erledigt wird. Beim Beispiel BitTorrent tragen die Peers am Rand des Netzes die Hauptlast, sie verteilen die Daten. Der Tracker ist nur ein Hilfsmittel zur Organisation, er ist nicht zentraler Punkt der Datenverarbeitung. Eine zentrale Eigenschaft von Peer-To-Peer-Systemen ist der notwendige Beitrag der Nutzer eines Dienstes zu seiner Funktionsweise.

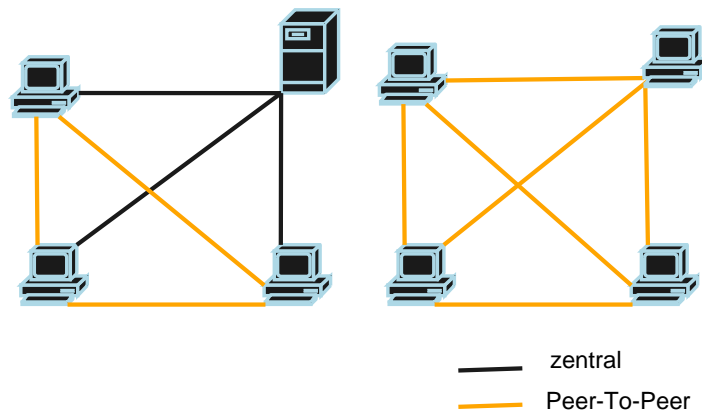


Abbildung 4.1: Organisationsformen in Peer-To-Peer-Netzen.
Links: Netz mit zentraler Komponente, Rechts: komplett dezentrales Netz.

Es gibt verschiedene Organisationsformen von Peer-To-Peer-Netzen (Abb. 4.1). Peer-To-Peer-Netze mit zentraler Komponente benötigen einen oder mehrere zentrale Punkte zur Verteilung von notwendigen Informationen über das Netz. Daneben gibt es komplett dezentrale Peer-To-Peer-Netze, in denen alle Teilnehmer komplett gleichberechtigt sind. Beide Varianten sollen mit verschiedenen Implementierungen und Organisationsformen vorgestellt werden.

4.1 Peer-To-Peer-Netze mit zentraler Komponente

Bei Peer-To-Peer-Netzen mit zentraler Komponente wird neben der direkten Kommunikation der Peers ein Teil der Funktionalität im Client-Server-Modell ausgeführt. Meistens besteht die Aufgabe der zentralen Bestandteile in der Verwaltung der Informationen über das Netz, also Bereitstellung von Kontakt- und Ressourceninformationen über andere Teilnehmer.

Um Teil des Peer-To-Peer-Netzes zu werden, stellt der neue Teilnehmer Kontakt zu einer der zentralen Komponenten her.

4.1.1 Napster

Napster wurde entworfen, um Dateien tauschen zu können. Das Napsternetz besteht neben den Peers aus Indexservern, denen jeder Peer die Dateien meldet, die er anbietet. Außerdem melden die Peers, mit welcher Übertragungsratesie angebunden sind. Die Indexserver stellen einen Suchdienst zur Verfügung, über den die Peers nach Dateien suchen können. Das Ergebnis der Suche ist eine Liste anderer Peers, die diese Dateien anbieten. Zusätzlich zu den Kontaktinformationen erhalten die Peers die Informationen über Netzanbindung, Dateigrößen und eine Gewichtung der Suchergebnisse. Die Dateiübertragung zwischen den Peers findet über ein spezielles Protokoll statt. Normalerweise baut der Peer, der die Datei

erhalten möchte, die TCP-Verbindung auf. Es gibt aber auch die Möglichkeit, dass der anbietende Peer die Verbindung aufbaut (push). Dazu wird die Vermittlung des Servers zur Herstellung der Verbindung benötigt. Jeder Peer entscheidet allerdings autonom, mit welchem anderen Peer er Kontakt aufnehmen will.

Neben Dateiübertragung bietet Napster auch dem IRC nachempfundene Kommunikationsmöglichkeiten. Diese Dienste arbeiten zentral über den Server. Jeder Napsternutzer muss sich beim Server registrieren und authentifizieren.

Man könnte das Napster-Protokoll [10] zur Implementierung einer Videostreaminglösung verwenden. Beim On-Demand-Streaming würde der Peer mit den Ursprungsdaten diese beim Napsterserver registrieren. Interessenten können diese dann herunterladen und unmittelbar abspielen. Allerdings stellt dies keinen Unterschied zum HTTP-Videostreaming dar, solange keine anderen Peers die Datei heruntergeladen haben und diese zum Upload anbieten. Beim Flash-Crowd-Syndrom entlastet dies den Anbieter nicht, da alle Teilnehmer gleichzeitig dieselbe Ressource von derselben Quelle anfordern. Livestreaming ist mit Napster allerdings nicht möglich, da im Napsterprotokoll Größeninformationen über Dateien übertragen werden und diese bei Livestreams veränderlich sind.

4.1.2 BitTorrent

4.1.2.1 Architektur des Peer-To-Peer Netzes

Das BitTorrent-Netz besteht aus einem Tracker und vielen Peers. Ein neuer Peer, der dem Netz beitreten will, findet Informationen über den Tracker sowie Prüfsummen in einer `.torrent`-Datei. Diese Datei hat sich der Nutzer über ein anderes Medium beschafft, üblicherweise von einem HTTP-Server. Der neue Peer verbindet sich mit dem Tracker und erhält eine zufällige Liste aus Adressen von Peers, die die gewünschte Ressource anbieten. Daraufhin kontaktiert der neue Peer verschiedene Peers aus der Liste. Fortschrittmeldungen über den Empfang von Datenblöcken festgelegter Größe werden von den Peers an den Tracker geschickt.

4.1.2.2 Auswahl der Peers

Allein der Tracker trifft die Entscheidung, welche Peers einander bekannt sind. Die einzige Entscheidung, die ein Peer treffen kann, ist, ob er einem anderen Peer Nutzdaten übertragen will. Dies geschieht über eine Zustandsaushandlung. Beide Peers haben 2 Zustandsbits: „interessiert“ oder „nicht interessiert“ und „gedrosselt“ oder „nicht gedrosselt“. Eine Übertragung findet nur statt, wenn der Empfänger im Zustand „interessiert“ und der Sender im Zustand „nicht gedrosselt“ ist. Damit haben die Peers eine signifikante Entscheidungsgewalt über die Form des Netzes.

Ein Peer sieht nicht zwangsläufig alle Teilnehmer im Netz, sondern nur die, deren Adressen ihm der Tracker mitgeteilt hat. Die offizielle BitTorrent-Implementierung teilt die Peers zufällig zu. Dieser Ansatz wurde von den Autoren in [11] gewählt, weil er robust ist.

4.1.2.3 Strategien für die Auswahl der Anforderung der Datenblöcke

BitTorrent zerteilt die Nutzdaten in Blöcke. Die derzeitigen Implementierungen verwenden standardmäßig eine Größe von 256 kB. Der Tracker kennt die Blöcke, die ein Peer verfügbar hat. Für eine einzelne Übertragung zwischen Peers werden diese Blöcke üblicherweise in Subblöcke zu 16 kB unterteilt.

Wichtigste Regel bei der Anforderung von Blöcken ist, dass Subblöcke, die zu einem Block gehören, von dem bereits andere Subblöcke angefordert wurden, ebenfalls angefordert werden. Dadurch wird bezweckt, dass Peers möglichst schnell komplette Blöcke besitzen und dies dem Tracker berichten können.

Eine weitere Strategie ist, dass jeder Peer zuerst Subblöcke aus dem Block anfordert, den die wenigsten der Peers anbieten, mit denen er verbunden ist. Damit wird bezweckt, dass der Peer etwas anbieten kann, was die mit ihm verbundenen Peers noch nicht heruntergeladen haben. Dies bewirkt eine gleichmäßige Verteilung der Blöcke auf alle Peers. Wenn ein Peer allerdings noch gar keine Blöcke anbieten kann, lädt er zufällig ausgewählte Blöcke herunter. Dadurch kann er relativ schnell etwas zum Download anbieten. Wenn der Peer zuerst einen Block herunterladen würde, den nur wenige der mit ihm verbundenen Peers anbieten, würde es lange dauern, bis er selbst etwas anbieten kann, weil üblicherweise bei den Peers, die die seltenen Blöcke anbieten, die Bandbreite knapp ist.

Im praktischen Einsatz von BitTorrent hat sich gezeigt, dass oft die letzten fehlenden Blöcke von einem Peer mit langsamer Leitung übertragen wurden. Deswegen wurde in BitTorrent der sogenannte „endgame mode“ implementiert. Dieser sorgt dafür, dass die betreffenden Subblöcke von allen Peers angefordert werden. Sobald ein Subblock eintrifft, werden die restlichen Requests für diesen Subblock abgebrochen.

Dass ein Peer einen Block anfordert, bedeutet nicht automatisch, dass das Gegenüber sofort damit anfängt, den Block zu übertragen. Stattdessen werden die Anfragen erst einmal in den Zustand „gedrosselt“ versetzt. Jeder Peer versetzt eine feste Anzahl Verbindungen zu anderen Peers in den Zustand „nicht gedrosselt“. Die Verbindungen, die in den Zustand „nicht gedrosselt“ gesetzt werden, wechseln. Die Zeitintervalle sind so gewählt, dass es möglich ist, dass die TCP-Verbindungen zwischen den Peers trotz TCP Slow Start die volle zur Verfügung stehende Bandbreite benutzen können.

Dabei werden die Peers zuerst ausgewählt, von denen die Downloadrate am höchsten ist. Damit auch Peers mit hoher Bandbreite, zu denen keine Daten übertragen werden, eine Chance haben, wird periodisch ein Uploadslot per Zufall an einen Peer vergeben. Dieser Slot wird dann für 3 Zeitintervalle fest an diesen Peer vergeben, damit die Chance besteht, hohe Datenraten von dem Peer empfangen zu können.

Durch die auf Zufall basierenden Algorithmen ist BitTorrent robust und erreicht hohe Datenraten. Allerdings garantiert BitTorrent zu keinem Zeitpunkt, dass angeforderte Daten auch geliefert werden. Für Videostreams, wo die Verfügbarkeit von Daten zum Abspielzeitpunkt ein entscheidendes Kriterium darstellt, ist BitTorrent daher nicht geeignet. BitTorrent ist dafür entworfen worden, den Datendurchsatz im BitTorrentnetz zu maximieren und damit an möglichst viele Teilnehmer möglichst schnell dieselben Daten zu verteilen. Das Kriterium, rechtzeitig Daten bereitzustellen, wurde im Systementwurf nicht berücksichtigt. Neben der fehlenden Garantie, dass bei Datenanforderung auch Daten übertragen werden, kann es auch durch den Systementwurf passieren, dass laufende Übertragungen abgebrochen werden und zufällig andere Teilnehmer mit Daten versorgt werden. Ein weiterer Punkt des Systementwurfs von

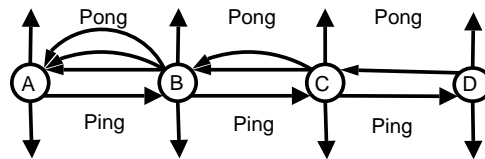


Abbildung 4.2: Gnutella: Ping

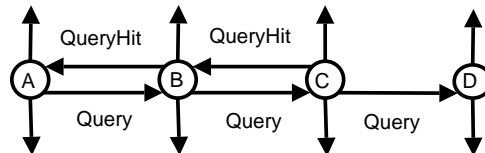


Abbildung 4.3: Gnutella: Query

BitTorrent spricht gegen die Eignung für Streaming: Beim Streaming werden immer Daten vom Sender zu dem oder den Empfängern versendet. Bei BitTorrent bestimmt dagegen die Datenrate in beiden Richtungen darüber, ob Daten übertragen werden oder nicht. Videostreaming sendet aber nur in einer Richtung zwischen zwei Peers.

4.2 Reine Peer-To-Peer-Systeme

Bei reinen Peer-To-Peer-Systemen sind alle Teilnehmer im Netzwerk gleichberechtigt. Kein Knoten ist wichtiger als ein anderer. Der Eintritt in das Peer-To-Peer-Netz geschieht durch Kontaktaufnahme zu einem beliebigen Netzteilnehmer. Die Adressen von bestehenden Teilnehmern werden über andere Medien übertragen. Möglichkeiten sind die Eingabe einer IP-Adresse durch den Nutzer, die Auslieferung von IP-Adresslisten mit der Software, Benutzung von Download-URLs oder Mechanismen zum automatischen Auffinden von Diensten [12].

4.2.1 Gnutella

Gnutella ist eines der frühen reinen Peer-To-Peer-Systeme, die für Filesharing entwickelt wurden. Gnutella ist kein eigenes Peer-To-Peer-Netz, sondern ein Protokoll. Es gibt verschiedene Implementierungen des Protokolls, teilweise mit Unterschieden im Protokoll. Die folgende Beschreibung bezieht sich auf die Protokollspezifikation in Version 0.4 [13].

Die Teilnehmer im Gnutellanetz heißen Servents. Die Aufgabe des Gnutella-Protokolls ist es, sowohl andere Netzteilnehmer zu finden als auch Dateien suchen zu können. Die Dateiübertragung findet nicht über das Gnutella-Protokoll statt, sondern direkt mittels HTTP zwischen dem Servent, der eine Datei anfordert und dem Servent, der die Datei anbietet. Die Aufgabe von Gnutella besteht also im Routing von Nachrichten und Finden anderer Servents.

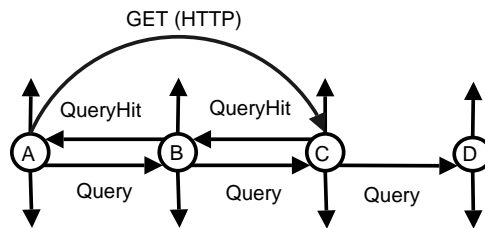


Abbildung 4.4: Gnutella: Get

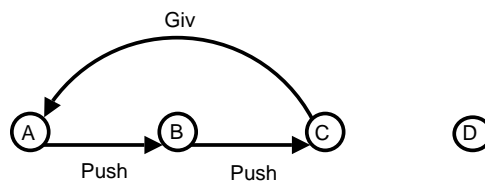


Abbildung 4.5: Gnutella: Push

Das Gnutella-Protokoll benutzt einen flutenden Ansatz, um anfragende Nachrichten im Netz zu routen. Das bedeutet, dass ein Servent Nachrichten zu allen benachbarten Servents weiterleitet, mit Ausnahme des Servents, von dem er die Nachricht bekommen hat. Antwortpakete werden immer an den Servent geschickt, von dem die dazu gehörende Anfrage empfangen wurde. Dadurch nehmen Antworten auf dem Rückweg denselben Pfad wie bei der Anfrage. Es setzt allerdings voraus, dass die Servents für eine gewisse Zeit speichern, welche Nachrichten sie schon weitergeleitet haben. Um Nachrichten nicht unendlich im Netz zirkulieren zu lassen, wurde ein Gültigkeitsmechanismus (TTL) wie bei IP-Paketen implementiert. Damit die Antworten den Anfragen zugeordnet werden können, gibt es Descriptor-IDs. Diese werden neben der Zuordnung für die Erkennung von Routingschleifen benutzt: Nachrichten mit bekannten Descriptor-IDs werden nicht weitergeleitet.

Das Gnutella-Protokoll definiert folgende Nachrichtentypen:

Ping: Der sendende Servent möchte aktiv andere Servents finden. Als Antwort werden Pong-Nachrichten erwartet.

Pong: Ein Servent, der eine Ping-Nachricht empfangen hat, sendet IP-Adresse und Port, über die er erreichbar ist. Zusätzlich werden statistische Daten wie Anzahl der angebotenen Dateien und Gesamtgröße aller Dateien übertragen. (Abbildung 4.2).

Query: Ein Servent sucht Dateien, die auf ein angegebenes Muster passen. Falls der Servent, der die Nachricht erhält (und ggf. weiterleitet) über Dateien verfügt, die zu dem Suchkriterium passen, sendet er eine QueryHit-Nachricht als Antwort.

QueryHit: In einer QueryHit-Nachricht werden Informationen über Dateien übertragen, die auf ein Suchkriterium passen. Dabei können mehrere passende Dateien eines Servents in einem QueryHit übertragen werden. Damit der anfragende Servent die Dateien direkt abfragen kann, werden IP-

Adresse und Port des anbietenden Servents mit übertragen. An der angegebenen Adresse ist dann ein HTTP-Server verfügbar, um Dateien zu übertragen. (Abbildungen 4.3 und 4.4)

Push: Wenn der Servent, der eine Datei anbietet, nicht erreicht werden kann, weil er sich hinter einer Firewall befindet, kann er aufgefordert werden, die Übertragung an den anfragenden Servent zu starten. Dazu schickt der anfragende Servent an den anbietenden Servent eine Push Nachricht. (Abbildung 4.5)

Durch den flutenden Routing-Algorithmus skaliert das Gnutella Netz sehr schlecht. Eine Ping-Nachricht in einem Netz mit n durch TTL-Begrenzung erreichbaren Servents löst n Pong-Nachrichten aus, die durch das Netz zum Sender geroutet werden. Der Overhead ist also beträchtlich. Andererseits ist das Netz durch den flutenden Ansatz robust. Wenn eine Nachricht innerhalb der Grenzen, die durch den TTL-Wert gesetzt sind, einen Peer erreichen kann, dann wird sie ihn erreichen. Ein einzelner unkooperativer Peer kann die Ausbreitung einer Anfrage nicht verhindern. Da die Antworten immer denselben Weg wie Anfragen nehmen, können allerdings Antworten blockiert werden.

Anders als BitTorrent stoppt Gnutella keine laufenden Datentransfers willkürlich und zieht auch keine Konsequenzen für die Auswahl der Partner aus der Datenrate. Daher ist es prinzipiell in die engere Kandidatenauswahl bei der Eignung für Videostreaming zu ziehen. Dank der Möglichkeiten von HTTP, Ressourcen teilweise zu übertragen, kann auch eine Live-Übertragung möglich gemacht werden. Wenn Videocodecs mit variabler Bitrate verwendet werden, ist es allerdings nicht möglich, bei Live-Übertragungen im Videostrom Sprünge (zurück) durchzuführen, da die Sprungziele¹ nicht im Medium selbst kodiert werden und Indizes aufgrund des Live-Charakters nicht im Voraus berechnet und übertragen werden können. Bei Ausfällen muss erst durch Flutung im ganzen Netz eine geeignete Ersatzquelle gefunden werden. Dies führt bei einem instabilen Netz innerhalb der durch das TTL-Feld gesetzten Grenzen zu vermehrtem Bandbreitenbedarf durch die zusätzlichen Nachrichten und dadurch möglicherweise zum Stocken der Videoübertragung.

4.2.2 Freenet

Freenet ist ein verteiltes Dateisystem, das ein Peer-To-Peer-Netz zum Auffinden, Speichern und Transport von Dateien benutzt. Entwickelt wird es vom Freenet-Projekt, dessen Zielsetzung in der Schaffung einer Möglichkeit, zuverlässig und anonym auf beliebige Daten zugreifen zu können, besteht. Das bedeutet, dass erstens der Nutzer, der Daten erstellt, zweitens der Nutzer, der Daten speichert, als auch drittens der Nutzer, der Daten abrufen, nicht gerichtsfest ermittelt werden können. Freenet soll alles bieten, was man benötigt, um Zensur zu verhindern. Dazu gehört auch, dass der Betreiber eines Freenet-Peers nicht wissen kann, welche Dokumente er speichert.

Im Freenet gibt es keine zentrale Komponente. Dateien werden durch den Hashwert ihres Inhalts identifiziert. Damit auch menschliche Nutzer Daten in Freenet finden können, gibt es noch einen Namensraum aus Hashwerten über Schlüsselwörter, mit denen die Hashwerte von Dateien gefunden werden können. Diese Schlüsselwörter sind nicht zur Funktionsweise von Freenet notwendig, sie stellen lediglich eine Erleichterung für die Nutzer dar.

¹Bei MPEG-Video können beispielsweise I-Frames als Sprungziele dienen

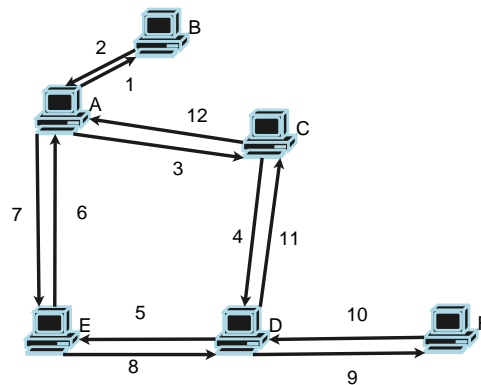


Abbildung 4.6: Routing in Freenet

Freenet benutzt ausschließlich die Schlüssel zum Routing von Nachrichten. Nachrichten können Anfragen nach Dokumenten sein oder die Aufforderung, Dokumente zu speichern. Wenn ein Knoten ein Dokument zu einem Schlüssel schon lokal vorrätig hat, kann er das Dokument an den anfragenden Knoten zurückschicken. Dadurch verteilen sich häufig angeforderte Dokumente über das Freenet. Bei jeder Nachricht (egal, ob Anfrage oder Antwort) entscheidet jeder Knoten lokal, wohin er diese weiterleitet. Jede Anfrage bekommt vom ursprünglichen Peer eine zufällige RequestID. Dabei wird die Routingentscheidung bei Anfragen anhand des Schlüsselwertes getroffen. Die Anfrage wird an den Knoten geschickt, von dem bekannt ist, dass er Dokumente mit nahem Schlüsselwert besitzt. Die Nähe zweier Schlüssel ist die numerische Differenz der Schlüsselwerte. Die passende Rückantwort wird an den Knoten geschickt, von dem die Anfrage mit der RequestID kam. Die Knoten müssen also für jede Anfrage Zustandsinformationen speichern.

Wenn ein Peer eine Nachricht mit einer bekannten RequestID bekommt, dann liegt eine Routing Schleife vor. In diesem Fall teilt er dies dem zweiten anfragenden Knoten mit. Dieser sucht dann in einem Backtrackingschritt bei anderen Knoten weiter. In Abbildung 4.6 soll der Routingalgorithmus veranschaulicht werden. Knoten A stellt eine Anfrage nach einem Schlüssel und leitet diese aufgrund der Informationen in seiner Routingtabelle an Knoten B. Das bedeutet, dass es bei Knoten B in der Vergangenheit andere Schlüssel gab, deren Schlüsselwert nah bei dem gesuchten Schlüssel liegt. B teilt A mit, dass er den Schlüssel nicht finden kann. Daraufhin sucht A bei C weiter. C wiederum leitet die Anfrage an D weiter, der sie an E schickt. E schickt die Anfrage an A, woraufhin A antwortet, dass er die Anfrage schon kennt und eine Schleife aufgetreten ist. Im Backtrackingschritt leitet E die Anfrage an D weiter. D entscheidet anhand seiner Routingtabelle, bei F zu suchen. F hat das gesuchte Dokument vorrätig und liefert es an D aus. D speichert es in seinem Cache und schickt es über C an A weiter. Ebenso wie D speichern C und A das Dokument in ihrem Cache.

Das Freenetrouting ist also eine Tiefensuche über das Netzwerk. Die Tiefensuche wird durch einen Hops-To-Live(HTL)-Zähler begrenzt. Dieser hat fast dieselbe Semantik wie das TTL-Feld im IP-Paketkopf. Eine Eigenschaft unterscheidet den HTL-Zähler allerdings vom TTL-Feld in IP: Wenn der HTL-Zähler den Wert 1 hat, kann der Knoten zufällig entscheiden, ob er die Nachricht weiterleitet oder nicht. Damit wird die Verfolgung von Nachrichten in einem Traceroute ähnlichen Verfahren verhindert.

Im Gegensatz zu Gnutella nehmen bei Freenet die Daten den gleichen Weg wie Anfragen. Auch dies ist in der Zielsetzung der Anonymität und Zensurresistenz begründet. Würden zwei Peers, die in der Netztopologie keine Nachbarn sind, direkt Daten miteinander tauschen, wäre die Anonymität nicht mehr gewährleistet. Daher müssen Daten und Anfragen den gleichen Transportmechanismus verwenden.

Freenet macht keinerlei Aussagen über Latenzzeiten für Datenübertragungen. Es gibt keinen Mechanismus, die Anzahl von Anfragen zu kontrollieren oder die Länge des maximal möglichen Weges effektiv zu begrenzen. Daneben gibt es Schwierigkeiten, wenn sich die Dateigröße ändert. Freenet ist für unveränderliche Dokumente konzipiert worden. Praktisch ist Freenet daher für Streaming nicht geeignet.

4.2.3 Routingalgorithmen für Overlay Networks

Die Skalierbarkeitsprobleme von Gnutella haben zahlreiche Wissenschaftler zur Entwicklung neuer Algorithmen angeregt. Das Ergebnis der Forschungen sind bessere Routingalgorithmen für Overlay Networks, die über eine physische Netzstruktur wie die des Internet eine virtuelle Netzwerkschicht legen. Über diese Overlay Networks werden Informationen verteilt, gesucht und gefunden.

Bei den hier vorgestellten Algorithmen werden den Nutzdaten Schlüssel in einem flachen Namensraum zugeteilt, so dass die Nutzdaten wie mit einer Hashtabelle gefunden werden können. Da der flache Namensraum sehr groß ist (üblicherweise 2^{160} Einträge), wird die daraus resultierende Hashtabelle auf mehrere Peers verteilt. Für die Konstruktion der verteilten Hasstabellen (*distributed hashtable*, DHT) gibt es eine Vielzahl an Algorithmen. Diese Algorithmen unterstützen mindestens eine Operation: zu einem gegebenen Schlüssel finden sie den dazu gehörenden Knoten.

4.2.3.1 Chord Algorithmus

Es wird angenommen, dass die Schlüssel für die verteilte Hashtabelle mit der SHA-1-Hashfunktion gewonnen werden. Es wird davon ausgegangen, dass Kollisionen in SHA-1 schwierig zu erzeugen sind². Die verteilte Hashtabelle enthält maximal 2^m Einträge.

Zum Finden eines Schlüssels benötigt Chord [15] lediglich $O(\log(N))$ Nachrichten bei N Peers. Jeder Peer kennt nur $O(\log(N))$ andere Peers. Dabei wird für jeden Schlüssel ein Knoten als Nachfolger definiert. Dieser Nachfolger ist für die Verwaltung des Schlüssels zuständig. Nachfolger für Schlüssel k ist der Knoten, der die kleinste Nummer modulo 2^m hat, die größer oder gleich k ist. Dazu ein Beispiel aus [15]: In einem Restklassenring der Größe 2^3 gebe es die Knoten 0, 1, 3 und die Schlüssel 1, 2, 6. Dann ist Knoten 1 für Schlüssel 1, Knoten 3 für Schlüssel 2 und Knoten 0 für Schlüssel 6 zuständig. Würde ein Knoten 7 hinzugefügt werden, würde die Verantwortlichkeit für Schlüssel 6 an Knoten 7 übergeben werden.

Das Auffinden von Schlüsseln geschieht über ein Routingprotokoll. Dazu existiert auf jedem Knoten ein Teil der Routingtabelle, der als *finger table* bezeichnet wird. Die *finger table* besitzt m Einträge.

²Inzwischen gibt es jedoch Vermutungen [14], dass es möglich sein könnte, ohne Brute-Force-Methoden Kollisionen in SHA-1 zu erzeugen.

Der i 'te Eintrag der *finger table* von Knoten n verweist auf den Nachfolgeknoten von $n + 2^{i-1}$. Dieser dort eingetragene Knoten ist für das Intervall $[n + 2^{i-1}, n + 2^i)$ zuständig. Wenn Knoten n nach einem Schlüssel k suchen möchte, sucht er in seiner *finger table* den Knoten n' , der für das Intervall zuständig ist, in dem k liegt. Falls Knoten n' nicht für den Schlüssel k zuständig ist, führt er nach gleichem Schema eine Suche in seiner *finger table* durch. Durch die Konstruktion der *finger table* ist gewährleistet, dass in einem stabilen System die Suche nach spätestens m Schritten beendet ist.

Über den gefundenen Schlüssel kann dann auf die gewünschte Information zugegriffen werden. Dabei kann der zum Schlüssel gespeicherte Wert Adressen enthalten, unter denen die gewünschte Information zu finden ist, oder auch die gewünschte Information selbst. Die Entscheidung, welche der beiden Möglichkeiten verwendet wird, ist eine Designentscheidung, die von der Anwendung, für die Chord benutzt wird, abhängt.

4.2.3.2 Kademia Algorithmus

Kademia [16] ist ebenfalls ein Algorithmus zur Konstruktion einer verteilten Hashtabelle. In der Hash-tabelle sind Schlüssel zu Daten und NodeIDs eingetragen. Zum Routing benutzt Kademia eine XOR-Metrik, die den Abstand zweier Einträge in der DHT bestimmt. Dabei ist der Abstand der numerische Wert des Ergebnisses der XOR-Operation, angewendet auf zwei Einträge. Dieser Abstand ist Basis für alle Routingentscheidungen in Kademia.

Schlüssel und Knoten haben bei Kademia eine Länge von 160 Bit. Die Routingtabelle besteht aus k -buckets, je einem k -bucket pro Bit Schlüssellänge. In diesen k -buckets wird eine begrenzte Anzahl Kontaktinformationen zu anderen Knoten gespeichert, sortiert nach Zeitpunkt des letzten Kontakts. Die k -buckets sind eine Datenstruktur mit Last-Recently-Seen-Ansatz. Sobald Kademia Kenntnis über einen neuen Netzteilnehmer erlangt, versucht es, diesen in den passenden k -bucket einzufügen. Dies funktioniert dann, wenn die passende k -bucket-Datenstruktur weniger als k Einträge hat. Wenn bereits k Einträge enthalten sind, testet Kademia die Erreichbarkeit des am längsten nicht gesehenen Eintrags. Ist dieser erreichbar, wandert der Eintrag nach oben in der Liste und der neue Eintrag wird nicht hinzugefügt. Ist der Knoten nicht erreichbar, wird er aus der k -bucket-Liste entfernt und der neue Teilnehmer eingefügt. Dadurch werden erreichbare Knoten niemals aus k -buckets verdrängt, was der Stabilität des Netzes zuträglich ist. Der Grund für diese Konstruktion ist die Erkenntnis der Kademia-Autoren, dass die Wahrscheinlichkeit, dass ein Knoten erreichbar bleibt, abhängig von seiner Uptime ist. Die Größe von k sollte so gewählt sein, dass es sehr unwahrscheinlich ist, dass alle k Knoten eines k -buckets gleichzeitig ausfallen. k ist gleichzeitig auch ein systemweiter Replikationsparameter.

Zum Routing benutzt Kademia vier verschiedene RPC-Aufrufe:

ping prüft ob ein Knoten noch erreichbar ist.

find_node(id) wird benutzt, um einen Knoten zu finden. Der Empfänger des RPC-Aufrufs liefert die k Knoten zurück, die id am nächsten sind. Wenn im passenden k -bucket weniger als k Einträge vorhanden sind, werden Einträge aus anderen k -buckets verwendet. Weniger als k Knoten werden nur zurückgeliefert, wenn der befragte Peer weniger als k andere Peers kennt.

find_value(id) verhält sich wie `find_node`, mit der Ausnahme, dass der gespeicherte Wert statt einer Knotenliste zurückgeliefert wird, wenn für die *id* ein `store`-Aufruf bei dem Knoten aufgerufen wurde. Sobald ein Knoten den Wert zurückgeliefert, endet die Prozedur.

store(id,value) speichert ein Schlüssel-Wert-Paar.

Das Finden eines Schlüssels verläuft nach folgendem Schema: Ein Knoten fragt die α Knoten nach dem Schlüssel, die in seinen k -buckets die geringste Distanz zum gesuchten Schlüssel haben. Aus den erhaltenen Antworten auf die RPC-Aufrufe sucht sich der Knoten wiederum die Knoten mit der geringsten Distanz heraus und fragt diese nach dem Schlüssel. Dabei ist α ein Parameter für die Parallelität im System. Kademia muss nicht warten, bis alle α Anfragen beantwortet wurden, sondern kann bereits weitere Knoten kontaktieren, wenn Antworten eingetroffen sind.

Zum Speichern eines Schlüssel-Wert-Paares sucht der speichernde Knoten die k nächsten Peers zum Schlüssel und führt bei diesen den `store`-Funktionsaufruf aus. Die gespeicherten Werte verfallen nach einer bestimmten Zeit, daher muss derselbe Schlüssel immer wieder regelmäßig im System gespeichert werden, um erhalten zu bleiben. Jeder Knoten speichert neben eigenen Werten (bei Filesharing: Dateien) auch Werte, die er mit `find_value` gefunden hat, im System, damit auch diese Werte gecached werden. Der Knoten, der den Wert erhalten hat, ruft `store` für diesen Wert bei den k nächsten Knoten auf.

4.2.3.3 Eignung für Streaming

Die vorgestellten Overlay-Techniken sind keine fertige Lösung für eine Anwendung, sondern bieten einen Baustein zur Realisierung. So wurde mittels Chord ein verteiltes Dateisystem [17] entworfen. Ein einfacher Lösungsansatz könnte Chord dafür benutzen, Dateien zu finden, indem der Node, der für den Schlüssel verantwortlich ist, die Daten der Datei ausliefert. Dabei wird zwar der Aufwand zur Suche verteilt, die Auslieferung der Datei allerdings nicht. Deshalb wurde ein komplexerer Ansatz gewählt. Dazu werden die Dateien in Blöcke zerteilt und ein Hashwert über den Inhalt der Blöcke wird als Key verwendet. Die Blöcke werden entlang des Routingpfades von Chord gecached. Damit eine Datei wieder zusammengefügt werden kann, existiert unter einem weiteren Schlüssel eine Metadatei ähnlich einem inode in Unix-Dateisystemen, wo beschrieben wird, in welchem Schlüssel die nötigen Informationen gesucht werden müssen, um die zerlegte Datei wieder zusammenzufügen. Diese Metadatei kann wiederum gecached werden.

Wie könnte nun eine entsprechende Lösung für Videostreaming aussehen? Beim Videostreaming ist die Hauptaufgabe die Vermeidung der übermäßigen Belastung einzelner Netzteilnehmer. Das heißt, dass das Peer-To-Peer-Netz nicht nur zum Auffinden der Daten genutzt werden muss, sondern auch zur Übertragung der Daten. Dabei könnte wie beim Chord-Filesystem der Videostrom in einzelne Blöcke zerteilt werden und die Zusammensetzung über Metadaten geregelt werden. Dies mag für On-Demand-Daten funktionieren, für Live-Ströme allerdings nicht, da die Metadaten ständig aktualisiert werden müssen, damit die Folgeblöcke gefunden werden können. Diese Informationen sind nur schlecht cachebar. Faktisch degeneriert damit das reine Peer-To-Peer-Netz zu einem semidezentralen Netz. Der Knoten, der für die Metadaten verantwortlich ist, übernimmt die Rolle des Trackers.

5 Konzept für ein Peer-To-Peer-System zur Übertragung von Streamingdaten

Wie in den Anforderungen in Kapitel 2 ausgeführt, soll die hier vorgeschlagene Lösung eines Peer-To-Peer Systems zur Übertragung von Streamingdaten gleichermaßen Livedaten als auch On-Demand Zugriff anbieten können.

5.1 Architektur

Im Fall einer Liveübertragung sollen alle Teilnehmer die gleichen Daten zur selben Zeit bekommen. Bei einer Peer-To-Peer-Struktur bietet es sich daher an, dass die Peers als Relay für andere Peers dienen und so die erforderliche Bandbreite am Ausgangspunkt verringern. Wenn jeder Peer die übertragenen Inhalte in einem Cache speichert, kann auch On-Demand-Zugriff angeboten werden. Dazu wird der Datenstrom in Access Units aufgeteilt, die adressiert und im Cache verwaltet werden. Damit Sprünge im Medienstrom möglich sind, muss das verwendete Medienformat folgende Eigenschaften erfüllen:

1. Bild und Ton müssen gemulplexed sein, das heißt zusammen übertragen werden.
2. Bild- und Tondaten aus Access Unit x dürfen keine Referenzen auf Access Unit y haben $\forall x \neq y$.
3. Access Units dürfen in beliebiger Reihenfolge abgespielt werden und bilden immer noch einen korrekt abspielbaren Medienstrom.

Zur Veranschaulichung: Eigenschaft 2 bedeutet im Fall von MPEG-Video, dass P- und B-Frames nur auf I-Frames verweisen dürfen, die in der selben Access Unit gespeichert sind. Analog gilt dies für die Tonkodierung.

Der Encoder zerlegt den kontinuierlichen Medienstrom in Access Units und stellt diese einem Peer auf geeignete Art und Weise atomar bereit, zum Beispiel, indem jede Access Unit in einer Datei gespeichert wird. Die atomare Bereitstellung einer jeden Access Unit ist wichtig, weil dann die Cacheverwaltung wesentlich vereinfacht werden kann. Die Access Units werden aufsteigend durchnummeriert, wobei der Access Unit 0 eine besondere Rolle zukommt – sie speichert den Header des Medienstroms. Alle weiteren Access Units enthalten nur noch abgeschlossene Mediendaten, die obige Eigenschaften erfüllen. Alle Access Units mit Mediendaten haben die gleiche zeitliche Länge. Durch diese Konstruktion ist gewährleistet, dass folgende Bedingungen erfüllt sind:

1. Zur Verfügung stehende Access Units ändern sich nicht mehr.

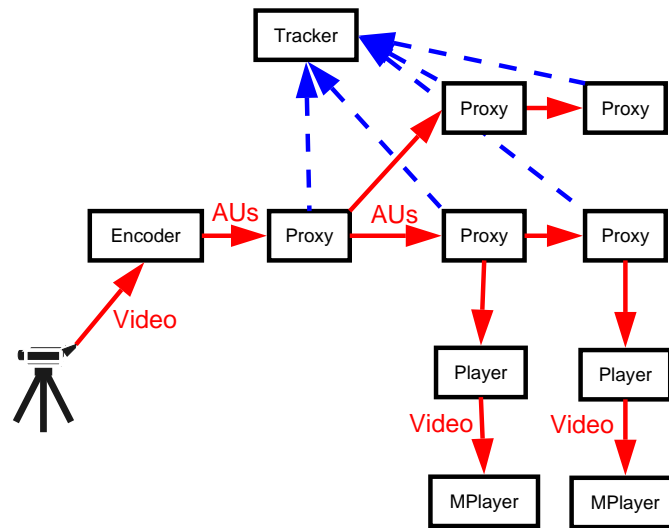


Abbildung 5.1: Datenfluss im konstruierten Peer-To-Peer-Netz

2. Über die Wahl der Access Units können direkte Sprünge im Medienstrom durchgeführt werden.
3. Access Unit x ist neuer als Access Unit $x - 1 \forall x > 0$.
4. Das Abspielen von Access Unit x dauert genauso lange wie das Abspielen von Access Unit $y \forall x > 0, y > 0$.

Die Einhaltung der obigen Bedingungen erlaubt folgende Schlussfolgerungen:

- Sobald Access Units zur Verfügung stehen, sind sie cachebar.
- Über die Ermittlung der Access Unit mit der höchsten Nummer kann an das Ende des Medienstroms gesprungen werden. Sei diese Access Unit z . Es ist gewährleistet, dass mindestens noch eine Access Unit abgespielt werden kann. Da alle Access Units die gleiche zeitliche Länge haben, liegt bei einer Liveübertragung nach Abspielen der Access Unit z eine neue Access Unit $z + 1$ auf dem Ausgangspunkt bereit, es sei denn, die Liveübertragung ist beendet.

Falls eine On-Demand-Übertragung stattfinden soll, fordert der Player Access Unit 0 an und anschließend in aufsteigender Reihenfolge Access Units mit Mediendaten. Bei einer Liveübertragung wird Access Unit 0 angefordert, und anschließend wird die letzte verfügbare Access Unit abgerufen. Ab dieser Position werden die Access Units in aufsteigender Reihenfolge abgespielt. Das Übertragungsprotokoll muss eine Möglichkeit zur Bestimmung der letzten gültigen Access Unit bieten.

Die Peers implementieren einen cachenden Proxy für das verwendete Übertragungsprotokoll. Zum Auffinden der Peers dient das Peer-To-Peer-Protokoll. Abbildung 5.1 zeigt den Datenfluss im Peer-To-Peer-Netz. Als Architektur für das Peer-To-Peer Netz wird ein semidezentraler Ansatz ähnlich wie bei BitTorrent gewählt. Die Peers fragen einen Tracker nach Adressen anderer Peers. Der Tracker liefert eine

Liste von möglichen Peers zurück. Dabei kann er für jeden Peer ein Ranking mitliefern. Dieses Ranking berechnet sich anhand der Access Units, die der Peer gespeichert hat, nach folgender Formel:

$$\sum_{i=current}^n \frac{1}{2^{i-current}} \cdot au_i \quad au_i = \begin{cases} 0: & \text{Access Unit } i \text{ nicht verfügbar} \\ 1: & \text{Access Unit } i \text{ verfügbar} \end{cases}$$

Dabei ist *current* die Access Unit, die der anfragende Peer als nächstes abspielen möchte. Durch die Konstruktion der Formel erzielt ein Peer, der *current* anbietet, ein höheres Ranking als ein Peer, der über alle Access Units außer *current* verfügt. Damit wird bezweckt, dass streamende Peers bevorzugt werden, weil dort die Wahrscheinlichkeit hoch ist, dass die darauf folgenden Access Units beim nächsten Zugriff ebenfalls vorhanden sein werden.

Sobald ein Peer eine Access Unit empfangen hat, meldet er dies dem Tracker. Sobald der Peer eine Access Unit löscht, teilt er dies ebenfalls dem Tracker mit. Ein Peer darf eine geforderte Access Unit allerdings an andere Peers ausliefert, auch wenn der Peer den Empfang noch nicht an den Tracker gemeldet hat. Damit wird die Verzögerungszeit durch die Proxykette verringert. Natürlich kann der Proxy die Daten nur so weit ausliefern, wie sie ihm selbst zur Verfügung stehen. Sofern auf den übergeordneten Peer gewartet werden muss, darf die Übertragung nicht abgebrochen werden, sondern es muss so lange gewartet werden, bis Daten vom übergeordneten Peer angekommen sind. Übertragungen dürfen nur beim Download abgebrochen werden, die Upload-Funktion muss beliebig lange warten können.

Der Tracker verwendet die Information über vorhandene Access Units bei den Peers lediglich für die Berechnung des Rankings. Die Rankings können den Peers als Orientierung bei der Auswahl geeigneter Peers dienen. Die Peers können sich aber auch nach anderen Kriterien als dem Ranking für geeignete Peers entscheiden, zum Beispiel anhand der IP-Adresse der Peers, um mögliche Lokalitäten in der Netzstruktur auszunutzen.

5.1.1 Organisation des Netzes

Zur Entscheidung über die Organisationsform des Netzes mussten die Optionen dezentrales und semi-dezentrales Peer-To-Peer-Netz unter Berücksichtigung des Einsatzzwecks abgewägt werden. Da Videowiedergabe zeitkritisch ist, kommt insbesondere der Zeit des Auffindens einer geeigneten Quelle ein besonderes Gewicht zu, weil der Nutzer eine Unterbrechung der Videowiedergabe bemerkt, sobald der Wiedergabepuffer leer ist. Das heißt, dass die Zeit zum Auffinden einer Quelle so gering wie möglich sein soll. Da im öffentlichen Internet üblicherweise keine Bandbreiten garantiert werden, können keine pauschalen Aussagen über Latenzzeiten zur Auffindung von geeigneten Peers gemacht werden. Bei einem semidezentralen Netz ist allerdings nur die Round-Trip-Time für Anfrage und Antwort zwischen Peer und Tracker zu betrachten. Bei einem reinen Peer-To-Peer-Netz müssen zur Ermittlung einer geeigneten Menge Peers in der Regel mehrere Nachrichten verschickt werden, so dass der Aufwand größer ist. Außerdem müssen die Nachrichten anderer Peers geroutet werden, was zusätzlichen Aufwand und damit Bandbreitenbedarf bedeutet. Ein reines Peer-To-Peer-Netz bietet dafür gegenüber einem semidezentralen Netz eine höhere Ausfallsicherheit, weil der Ausfall einer einzelnen Komponente nur Teile des Netzes betrifft. Beim semidezentralen Netz führt der Ausfall des Trackers zum Nichtfunktionieren des Netzes.

Natürlich muss mit Bandbreite sparsam umgegangen werden. Dies ist insbesondere von großer Bedeutung, weil ein großer Teil der Bandbreite für die Nutzlast, also die Videodaten, benötigt wird. Außerdem müssen die Nutzdaten rechtzeitig vorhanden sein. Plötzliche Zusatzbelastung durch Verwaltung des Netzes, wie sie beispielsweise durch Broadcaststürme bei Gnutella auftreten können, können die Dienstqualität erheblich verringern. Bei einem semidezentralen Netz wird jeder Peer nur durch den Overhead, den er selbst für das Auffinden anderer Peers benötigt, belastet. Bei einem reinen Peer-To-Peer-Netz hat jeder Peer den Overhead für sich und zusätzlich den für andere Netzteilnehmer zu verarbeiten. Beim semidezentralen Netz summiert sich der Trafficoverhead beim Tracker. Untersuchungen zur Trackerperformance und inwieweit die semidezentrale Lösung skalieren kann, folgen in Kapitel 7.3. Dort wird gezeigt, dass die Skalierbarkeit der semidezentralen Lösung für die Erfüllung der Aufgabenstellung ausreicht.

In Kapitel 4.2.3.3 wurde außerdem erläutert, dass ein reines Peer-To-Peer-Netz leicht zu einem semidezentralen Netz degenerieren kann, wenn die Mechanismen von verteilten Dateisystemen für Videostreaming übernommen werden.

5.1.2 Übertragungsprotokoll für die Mediendaten

Als Übertragungsprotokoll für die Mediendaten wurde HTTP [1] gewählt. Die Peers implementieren also einen HTTP-Proxy. Die Wahl fiel auf HTTP, zum einen, weil es gut erforscht und in zahlreichen Implementierungen verfügbar ist, zum anderen muss wieder der Blick auf den Einsatzzweck und die Alternativen gerichtet werden. Als Alternative käme zum Beispiel RTSP/RTP in Frage. Bei der Benutzung von RTP wäre allerdings die Gewährleistung der Unveränderlichkeit (Bedingung 1, Seite 24) schwer erreichbar, weil RTP keinen Mechanismus zur Wiederholung der Übertragung verlorener Pakete vorgesehen hat. Dies ist aber wünschenswert, um zu verhindern, dass sich lokale Übertragungsfehler in der Proxykette fortpflanzen. Die Fehler pflanzen sich nicht nur fort, was zu einer Qualitätsverschlechterung führen würde, sondern sind auch noch dauerhaft, wenn nicht ein Synchronisationsprotokoll durchgeführt würde, welches Übertragungsfehler korrigiert. Dieses Synchronisationsprotokoll sollte sinnvollerweise erst nach der Übertragung, oder wenn der Benutzer die Übertragung unterbricht und keine anderen Peers Daten anfordern, durchgeführt werden, da sonst die dazu notwendige Bandbreite möglicherweise nicht zur Verfügung steht und zu erneuten Übertragungsfehlern bei den Nutzdaten führen würde. Wenn der Benutzer die gewünschte Übertragung gesehen hat, beendet er aber wahrscheinlich die Software, so dass die Korrekturübertragung nutzlos ist.

Natürlich kann HTTP keine Paketverluste im Netz verhindern. Allerdings garantiert HTTP die korrekte und vollständige Übertragung der Daten, sofern dies möglich ist. Damit eine einzelne Blockade der Übertragung in der Proxykette nicht zur Blockade aller nachfolgenden Proxies führt, wird nach Ablauf eines Timeouts ein anderer Proxy zur Übertragung gesucht. Der Empfangspuffer im Player muss groß genug sein, um Timeouts überbrücken zu können. Ansonsten bekommt der Benutzer eine Störung im Netz mit. Trotz der möglichen Blockade durch TCP in der Proxykette ist diese Konstruktion vorteilhaft. Die Blockade wirkt sich nur so lange aus, wie Daten liefernder und empfangender Peer gleichzeitig dieselbe Position im Videostrom verarbeiten. Das ist bei Liveübertragungen der Fall. Bei On-Demand-Übertragungen besteht die Möglichkeit, dass der sendende Peer die Daten bereits in seinem Cache hat und die Datenübertragung daher nicht von anderen Peers abhängig ist.

5.1.3 Adressierung

Das Peer-To-Peer-Netz benutzt eine zweistufige Adressierung. Die erste Stufe gibt die Ressource an, die nächste Stufe die Access Unit. Die erste Stufe wird über eine Basis-URL realisiert, an die einfach der Name der Access Unit angehängt wird. Beispiel: Aus Ressource `http://example.com/streaming/hi/` und Access Unit 42 wird `http://example.com/streaming/hi/42`. Die geforderte Möglichkeit, Übertragungen in verschiedenen Qualitäten anbieten zu können, wird über die Wahl geeigneter Basis-URLs realisiert. Die entsprechende Basis-URL für niedrige Qualität könnte `http://example.com/streaming/lo/` lauten.

HTTP dient auch zur Bestimmung der letzten verfügbaren Access Unit. Dazu konstruiert der Peer eine URL aus der Basis-URL und einem speziellen Namen, zum Beispiel `latest`. Die Antwort darauf ist keine Access Unit, sondern ein HTTP-Redirect auf die aktuelle letzte Access Unit. HTTP-Redirects werden nicht gecached, so dass diese Anfrage bis zur Quelle durchgeleitet wird. Wenn dies ein Problem für die Belastung der Quelle darstellt, kann die Antwort allerdings auch problemlos für die Abspieldauer einer Access Unit auf den benutzten Proxies gespeichert werden. Der Preis, der für die Entlastung der Quelle bezahlt werden muss, ist der mögliche Verweis auf die vorletzte Access Unit.

5.2 Beschreibung des Trackerprotokolls

Neben dem Datenübertragungsprotokoll HTTP wird noch ein Protokoll zur Kommunikation mit dem Tracker benötigt. Da der Tracker ein wichtiger Faktor für die Skalierbarkeit des Peer-To-Peer-Netzes ist, soll das Kommunikationsprotokoll mit dem Tracker möglichst einfach sein. Daher wird ein eigenes einfaches Zeilenprotokoll verwendet. Dieses soll in diesem Kapitel beschrieben werden. Das Protokoll verwendet Klartextnachrichten in einem festen Format, so dass es sich mit wenig Aufwand in Scriptsprachen, aber auch in C implementieren lässt. Dabei werden folgende Nachrichten definiert: `FINDPROXY`, `FOUNDPROXY`, `DROPPROXY`, `AVAILABLE`, `DROP` und `PROVIDES`.

Eine Anfrage an den Tracker hat folgende Form: ein Peer sendet einen Befehl an den Tracker. Die Parameter für den Befehl werden durch Leerzeichen getrennt angehängt. Das Ende des Befehls wird durch das Zeilenendezeichen (`\n`) signalisiert. Die Antwort ist ein Antwortcode, der durch Parameter bzw. Erläuterungen ergänzt wird. Wenn die Antwort länger als eine Zeile ist, wird die Anzahl Zeilen als Parameter übergeben. Das Ende der Antwort wird durch `\n` signalisiert. Anschließend folgt wenn nötig die vereinbarte Anzahl Zeilen. Grundsätzlich werden Daten in ihrer Textrepräsentation übertragen. Bei Zahlen kann die entsprechende Darstellung beispielsweise durch die Funktion `sprintf` der C-Bibliothek erzeugt werden.

FINDPROXY teilt dem Tracker mit, dass ein Peer andere Peers finden möchte. Parameter sind die Basis-URL des abzuspielenden Videos, die Anzahl der Peers, die der Peer maximal bedienen kann und die Position, ab der er das Video abspielen möchte. Der Parameter für die Position wird für die Ermittlung des Rankings durch den Tracker verwendet. Als Antwort erhält der Peer eine Liste von IP-Adress-Port-Paaren mit einem Ranking.

C⇒T FINDPROXY \$URL \$IP_ADR \$PORT \$CLIENTS \$START_AU \$NR_PEERS
\$URL ist die Basis-URL der Ressource. \$IP_ADR ist die IP-Adresse des anfragenden Peers. Entsprechendes gilt für \$PORT. \$CLIENTS gibt die Anzahl der Peers an, die dieser Peer zu versorgen bereit ist. \$START_AU gibt die Access Unit an, ab der der Peer Material abspielen möchte. \$NR_PEERS gibt an, wie lang die Liste der ausgelieferten Peers maximal sein darf.

C⇐T 305 \$NR_PEERS proxies\n \$PEERLIST
\$NR_PEERS ist die Anzahl Peers. \$PEERLIST sind \$NR_PEERS Zeilen mit folgendem Format: \$IP_ADR:\$PORT \$RANK

C⇐T 404 NOT FOUND
Der Tracker verwaltet die in \$URL angegebene Ressource nicht.

C⇐T 500 0 proxies
Wenn keine Kandidaten als Peer gefunden werden konnten, wird dies durch den Code 500 signalisiert.

FOUNDPROXY teilt dem Tracker die IP-Adresse des Peers mit, die der Peer als Quelle ausgesucht hat.

C⇒T FOUNDPROXY \$URL \$IP_ADR \$PORT
\$URL ist die Basis-URL der Ressource, für die der Peer mit Adresse \$IP_ADR:\$PORT als Quelle verwendet wird.

C⇐T 200 OK
Die Änderung wurde im Tracker vermerkt

C⇐T 403 FORBIDDEN
Die übergebene Adresse darf nicht als Proxy verwendet werden.

DROPPROXY teilt dem Tracker mit, dass der angegebene Peer nicht mehr als Quelle verwendet wird.

C⇒T DROPPROXY \$URL \$IP_ADR \$PORT
\$IP_ADR:\$PORT wird nicht mehr als Quelle benutzt.

C⇐T 200 OK
Die Änderung wurde im Tracker vermerkt

C⇐T 404 NOT FOUND
Die angegebene IP-Adresse war kein Proxy des Peers

AVAILABLE teilt dem Tracker mit, dass der Peer den angegebenen Bereich Access Units vorrätig hat. Der Bereich wird durch die Nummer der ersten und letzten Access Unit angegeben. Wenn der Peer dem Tracker nicht zusammenhängende Bereiche mitteilen möchte, muss er mehrere AVAILABLE-Befehle ausführen.

C⇒T AVAILABLE \$BEGIN \$END
\$BEGIN und \$END geben das abgeschlossene Intervall im Access Unit-Nummernraum an, das der Peer vorrätig hat. Das Intervall darf bereits gemeldete Access Unit Bereiche überschneiden.

C⇐T 100 OK
Die Änderung wurde im Tracker vermerkt

C⇐T 416 REQUEST RANGE NOT SATISFIABLE
Die Bereichsangabe wurde vom Tracker nicht akzeptiert

DROP teilt dem Tracker mit, dass der Peer einen Bereich aus Access Units nicht mehr vorrätig hat. Die Bereichsangaben haben dasselbe Format wie bei der AVAILABLE-Nachricht.

C⇒T DROP \$BEGIN \$END
\$BEGIN und \$END geben das abgeschlossene Intervall im Access-Unit-Nummernraum an, das der Peer aus seinem Cache entfernt hat. Das Intervall darf auch Bereiche überschneiden, die der Peer gar nicht als vorrätig gemeldet hat.

C⇐T 100 OK
Die Änderung wurde im Tracker vermerkt

C⇐T 416 REQUEST RANGE NOT SATISFIABLE
Die Bereichsangabe wurde vom Tracker nicht akzeptiert

PROVIDES wird von dem bereitstellenden Peer aufgerufen und teilt dem Tracker mit, dass der Peer die entsprechende Ressource mit entsprechender URL anbietet. Anschließend ruft der Peer AVAILABLE auf, sobald neue Access Units zur Verfügung stehen.

C⇒T PROVIDES \$URL \$IP_ADR \$PORT \$CLIENTS \$CREDENTIALS
\$URL ist die Basis-URL, unter der das Material verfügbar ist. \$IP_ADR, \$PORT und \$CLIENTS haben die gleiche Semantik wie in FINDPROXY. \$CREDENTIALS ist eine Textfolge, die den Anbieter identifiziert. Das Format der Textfolge ist nicht weiter spezifiziert, sondern hängt von der Implementierung im Tracker ab.

C⇐T 200 OK
Die Ressource wurde registriert.

C⇐T 401 NOT AUTHORIZED
Die Daten im Feld \$CREDENTIALS berechtigen nicht zur Registrierung eines URL.

C⇐T 409 CONFLICT
Die zu registrierende Basis-URL ist dem Tracker schon bekannt.

5.3 Unterstützung von DSL-Nutzern

DSL-Netze stellen zum Zeitpunkt der Erstellung dieser Arbeit einen kostengünstigen Breitbandzugang für weite Teile der Bevölkerung dar. Es sind damit Datenraten im Bereich von mehreren hundert bis wenigen tausend Kilobit pro Sekunde Downstream erreichbar. Beim Upstream ist es technisch ebenfalls möglich, vergleichbare Datenraten wie beim Downstream zu erzielen, jedoch verfügen fast alle für Privathaushalte bezahlbaren Angebote über wesentlich geringere Bandbreiten beim Upload, üblicherweise in der Bandbreitenspanne zwischen 128 und 512 Kilobit pro Sekunde. Dies hat für die Implementierung eines Peer-To-Peer-Netzes weitreichende Auswirkungen. Da die Uploadbandbreite in der Regel nicht ausreicht, um selbst das niedrig aufgelöste Material in Echtzeit *einem* anderen Teilnehmer im Netz zur

Verfügung zu stellen, müssen entsprechende Maßnahmen getroffen werden, um das Peer-To-Peer-Netz trotzdem mit diesen Anschlüssen funktionieren zu lassen. Es ist notwendig, dass sich ein Peer die Nutzdaten von mehreren anderen Peers gleichzeitig herunterlädt, da diese voraussichtlich ebenfalls nur über einen DSL-Anschluss verfügen. Das Problem bei dieser Lösung ist die längere Latenzzeit beim Start. Es muss erst eine komplette Access Unit von einem Peer heruntergeladen werden, der diese aber nicht schnell genug liefern kann, um sofort mit dem Abspielen beginnen zu können. Die Proxysoftware muss daher mit parallelen Downloads und mehreren Downloadquellen umgehen können. Im Trackerprotokoll gibt es deshalb die Kommandos FOUNDPROXY und DROPPROXY.

5.4 Kostenabschätzung für den Tracker

Um die Leistungsgrenze eines performanten Trackers besser abschätzen zu können, ist eine Kostenabschätzung der einzelnen Operationen sinnvoll. Dazu sind Angaben über eine Implementierung nötig. In Bezugnahme auf den Einsatzzweck können Annahmen gemacht werden, die eine effektive Implementierung ermöglichen. Die Information, ob eine Access Unit verfügbar ist, ist binär, also kann die Menge der vorhandenen Access Units über ein Bitfeld repräsentiert werden. Dabei ist eine sinnvolle Anfangsgröße zu wählen, z.B. 4000 Bit bei 4 kByte Seitengröße. Bei einer Access Unit-Länge von 10 Sekunden können damit 170 Minuten und 40 Sekunden Material adressiert werden – für eine Vorlesung ist das ausreichend. Wenn das nicht genügt, kann über ein (Mehrfach-) Indirektionsverfahren wie bei Inodes im ext2-Dateisystem zusätzlicher Platz für Bitfelder geschaffen werden. Mit den oben gewählten Parametern können auf einer Speicherseite 8 Instanzen der Datenstruktur gespeichert werden. Zur Veranschaulichung dient Abbildung 5.2.

Die Operationen AVAILABLE und DROP lassen sich damit effizient implementieren. Pro Access Unit ist eine Schreiboperation für das Setzen oder Löschen des Bits notwendig, wobei Operationen bis zur Registerbreite zusammengefasst werden können. Zusätzlich sind noch bis zu 3 Leseoperationen nötig, um die Indirektionsstufen zu traversieren.

Für die Operation FINDPROXY muss jede Access Unit der gewählten Peer-Menge betrachtet und das Ranking berechnet werden. Dazu muss für jede Access Unit eine Addition und eine Multiplikation durchgeführt werden. Die Anzahl der Additionen und Multiplikationen für FINDPROXY wird damit also durch $O(au \cdot p)$ begrenzt. Man kann dies weiter begrenzen, wenn man die Genauigkeit verringert, indem man die Zahl der betrachteten Access Units begrenzt. Wenn jeweils die besten p Peers zurückgeliefert werden, erhöht sich der Berechnungsaufwand auf $O(au \cdot p \cdot P)$ oder $O(au \cdot P \log P)$, je nachdem, ob sortiert wird, oder die besten p Adressen (sequentiell) herausgesucht werden. P ist dabei die Zahl aller verbundenen Peers, au die Zahl der Access Units. Allerdings darf der Tracker nur Peers zurückliefern, die noch Kapazität für Uploads haben. Die Anzahl der notwendigen Speicherzugriffe verhält sich analog. Falls der Tracker kein Ranking erstellt, wird die Zahl der Operationen durch $O(p)$ begrenzt. Wenn die Anzahl der ausgelieferten Adressen begrenzt wird, kann sogar konstante Laufzeit erreicht werden.

Für jeden Peer wird sowohl eine Liste an Peers, die er als Proxy benutzt, als auch eine Liste an Peers, für die er Proxy ist, gespeichert. Die Anzahl Peers, für die ein Peer Proxy sein kann, ist signifikant kleiner als die Gesamtzahl an Peers und wird als konstant betrachtet. Folglich ist auch der Aufwand für die

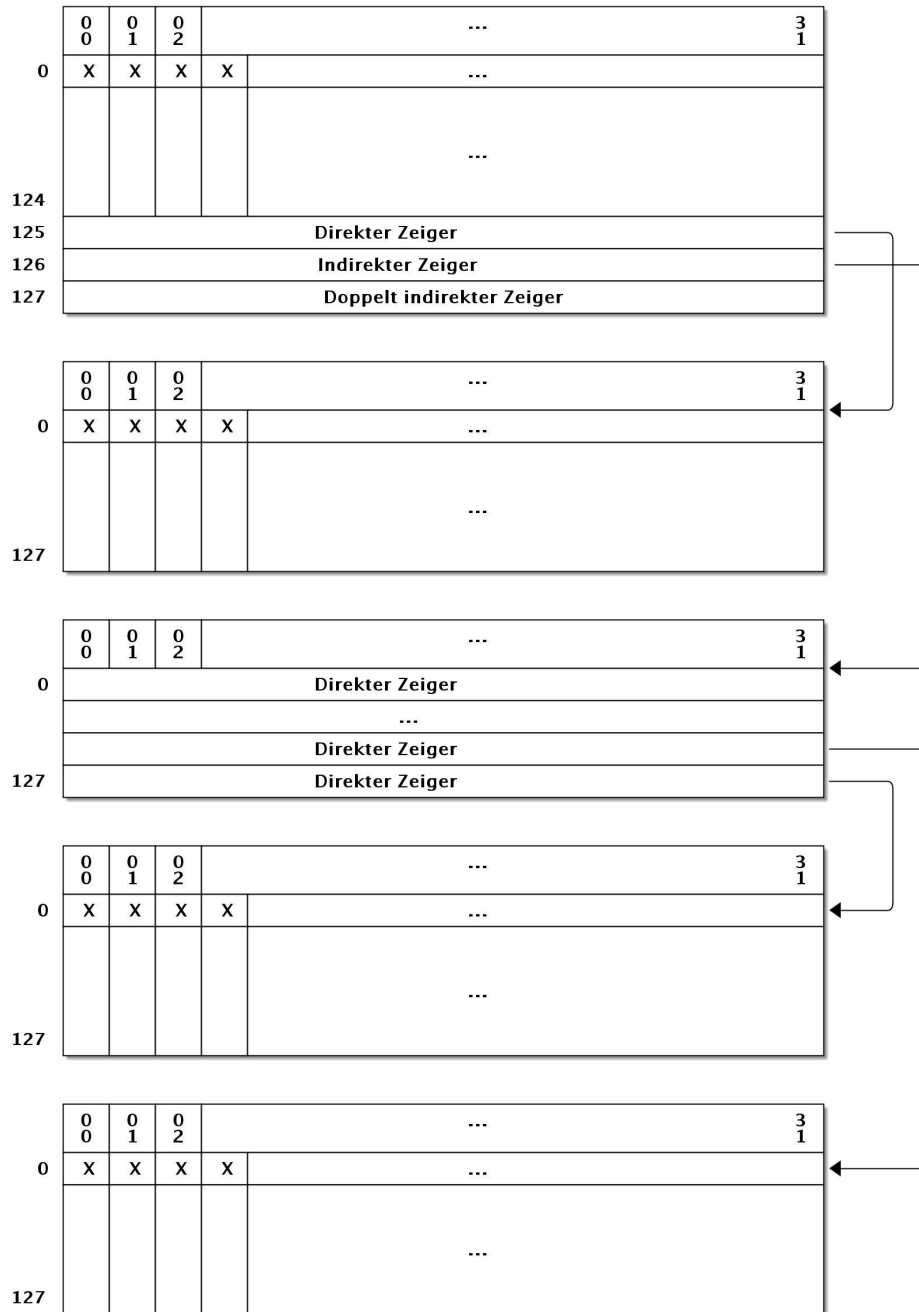


Abbildung 5.2: Datenstruktur zur Speicherung des Vorhandenseins der Access Units

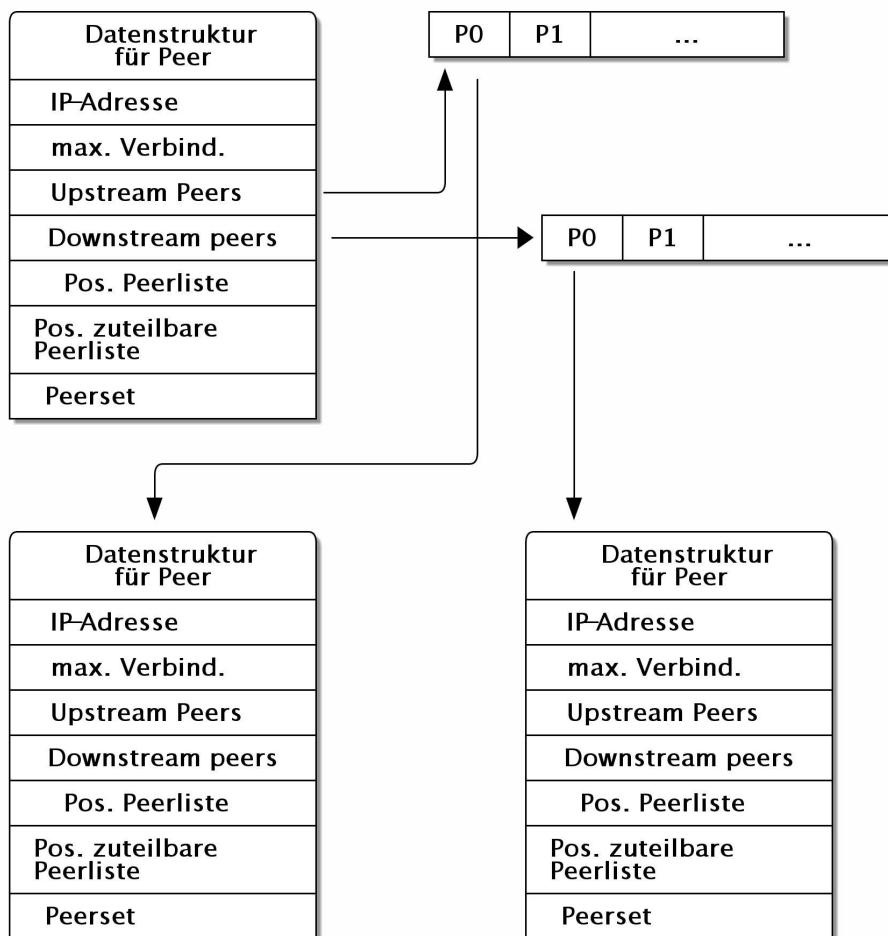


Abbildung 5.3: Datenstruktur zur Speicherung der Verbindungen zwischen den Peers

Operationen FOUNDPROXY und DROPPROXY konstant. Diese Vereinfachung ist zulässig, weil jeder Peer nur eine feste Maximalanzahl anderer Peers mit Daten versorgen kann – die Netzbandbreite stellt dort ein festes Limit dar¹. Eine einfache Datenstruktur für die Verwaltung dieser Liste sind Zeigerfelder wie sie in Abbildung 5.3 zu sehen sind.

Natürlich muss der Tracker auch die Verbindungen zu den einzelnen Peers speichern. Dazu müssen Einfüge- und Löschoptionen auf der Menge der Peers durchgeführt werden. Bei der Organisation als Zeigerfeld könnte folgender Mechanismus verwendet werden: Sobald sich ein neuer Peer mit dem Tracker verbindet, wird dieser als letztes Element im Feld eingetragen. Wenn ein Peer die Verbindung trennt, wird sein Platz im Feld mit der Adresse des letzten Eintrags ersetzt und die Anzahl Peers ver-

¹Der Tracker kann diese Maximalanzahl sogar erzwingen, da er die Kontrolle darüber hat, wem er welche Kontaktadressen ausliefert.

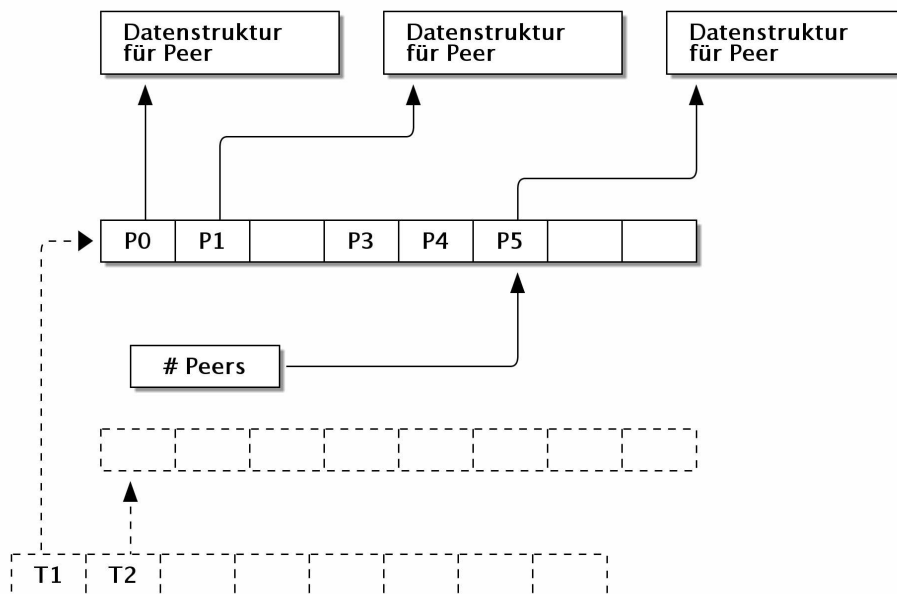


Abbildung 5.4: Datenstruktur zur Speicherung der Verbindungen zu den Peers

ringert. In Abbildung 5.4 ist diese Datenstruktur dargestellt. Bei der Implementierung muss abgewägt werden, ob es sich lohnt, die Menge in mehrere Blöcke fester Größe zu unterteilen, die über eine Indextabelle zusammengehalten werden. In Abbildung 5.4 ist die Indextabelle mit unterbrochenen Linien dargestellt. Die Parameter zur Entscheidungsfindung sind Codekomplexität und längere Zugriffszeiten gegen Speicherverbrauch. Selbst bei hohen Teilnehmerzahlen ist der verschwendete Speicher nach aktuellen Maßstäben vergleichsweise gering, zumal nur zwei Instanzen der Datenstruktur existieren und Speicherseiten bei Betriebssystemen mit virtuellem Speicher nur dann physikalischer Speicher zugeteilt wird, wenn darauf zugegriffen wurde. Hierzu eine Beispielrechnung: wenn maximal 100000 Teilnehmer bedient werden sollen, werden 400 kB Adressraum für jede Instanz der Datenstruktur benötigt. Der notwendige Speicher im Betriebssystem für 100000 TCP-Sockets ist wesentlich größer.

Eine Instanz der Datenstruktur enthält alle Peers, die mit dem Tracker verbunden sind. Die andere enthält alle Peers, bei denen noch nicht alle Uploadslots vergeben sind. Zum schnelleren Zugriff enthält die Datenstruktur jedes Peers die Nummer des Eintrages im jeweiligen Zeigerfeld. Diese muss ebenfalls mit angepasst werden, wenn die Zeiger an andere Positionen verschoben werden. Der Eintrag in der Instanz für die Peers mit Uploadkapazität muss gegebenenfalls in der Funktion FOUNDPROXY oder DROPPROXY ebenfalls angepasst werden.

In Tabelle 5.1 sind die Kosten der einzelnen Operationen zusammengefasst.

Der Speicherverbrauch im Tracker wird ebenfalls durch $O(au \cdot P)$ begrenzt. Dies ist leicht nachvollziehbar. Für jeden verbundenen Peer gibt es eine Liste der Access Units mit Größe $O(au)$. Weiterhin wird

Operation	Kosten
AVAILABLE	$O(1)$
DROP	$O(1)$
FINDPROXY mit Ranking	$O(au \cdot p)$
FINDPROXY ohne Ranking	$O(p)$
FINDPROXY ohne Ranking (beschränkt)	$O(1)$
FOUNDPROXY	$O(p)$
FOUNDPROXY (beschränkt)	$O(1)$
DROPPROXY	$O(1)$
Verbinden	$O(1)$
Trennen	$O(1)$

Tabelle 5.1: Kosten der Operationen im idealen Tracker

für die Speicherung der Datenstrukturen der Peers Speicher gebraucht, also $O(P)$. Der Speicher für die Verbindungen zwischen den Peers wird als Konstante für jeden Peer mit aufgenommen. Damit ist obige Aussage belegt. Zum Speicherverbrauch im Tracker kommt noch der Speicherverbrauch im Betriebssystem für die TCP-Verbindungen hinzu. Da dies aber von jedem Betriebssystem – teilweise sogar von Version zu Version – unterschiedlich gehandhabt wird, soll dies hier nicht weiter betrachtet werden. Weiterhin kommt der Speicherverbrauch der Laufzeitumgebung hinzu.

6 Implementierung

Zum Nachweis der Funktionsfähigkeit des Konzepts wurde ein Prototyp implementiert. Das Peer-To-Peer-Netz besteht aus drei unabhängigen Komponenten: *Tracker*, *Proxy* und *mplayer-glu*e. Dazu kommen noch die Komponenten für Videoeingabe, -komprimierung und -darstellung. Die Videokomponenten wurden entweder in [18] entwickelt oder sind frei verfügbare Open-Source-Programme.

6.1 Tracker

Der Prototyp für den Tracker wurde in Perl implementiert und kommuniziert über das in Kapitel 5.2 spezifizierte Protokoll mit den Peers. Zur Datenspeicherung verwendet die Implementierung eine PostgreSQL-Datenbank [19]. Der Tracker ist eine Art Application Server, der die Anfragen der Peers in Datenbankabfragen umsetzt. Als Servermodell wird für den Tracker das Einprozessmodell verwendet, das heißt, dass ein Thread alle verbundenen Peers bedient. Dazu wurde das `Net::Server::Multiplex`-Modul aus dem CPAN [20] verwendet. Dieses benutzt intern den `select`-Systemruf. Die entsprechenden Begrenzungen für die Skalierbarkeit werden durch diese Implementierung vorgegeben. Allerdings ist es möglich, mehrere Trackerinstanzen zu starten, die auf derselben Datenbank operieren, um die Beschränkungen durch `select` zu umgehen. Das verwendete Datenmodell ist in Abbildung 6.1 zu sehen.

6.2 Proxy

Der Proxy implementiert das HTTP-Protokoll und kommuniziert zur Verwaltung des Peer-To-Peer-Netzes mit dem Tracker. Die Implementierung wurde in Python [21] realisiert und wurde so gestaltet, dass sie unter Unix und unter Windows funktioniert. Der Proxy verwendet Threads zum gleichzeitigen Download und zur Bedienung anderer Peers, weil damit die Implementierung einfacher realisiert werden konnte. Threads verursachen Schwierigkeiten, wenn viele Clients parallel von einem Server bedient werden müssen. Für die Aufgabenstellung stellt dies aber kein Problem dar, da davon ausgegangen werden kann, dass ein Peer nicht mehr als 50 andere Peers mit Videodaten versorgen wird (2 MBit/s Videostrom, 100 MBit/s Netzanschluß).

Die Annahme von Requests wird über die Klasse `ThreadingHTTPServer` realisiert, die von der in Python mitgelieferten Klasse `ThreadingTCPServer` erbt. Diese benötigt eine weitere Klasse zur Abarbeitung der Requests. Diese Klasse ist `ConstrainedHTTPRequestHandler`, die die HTTP-Anfragen verarbeitet. In der Methode `do_GET` wird ein Großteil der Entscheidungen über den weiteren Verlauf des Requests gefällt. Zuerst wird geprüft, ob noch ein Uploadslot verfügbar ist. Wenn nicht, wird der HTTP Fehler-

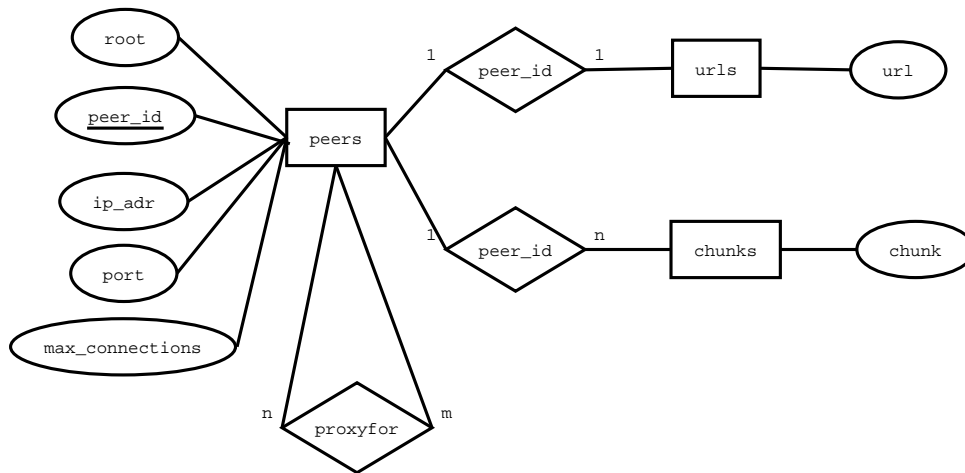


Abbildung 6.1: Datenmodell für den Tracker

code 503 (Service Unavailable) zurückgeliefert und die Verbindung beendet. Ansonsten wird getestet, ob die angeforderte Access Unit schon vorrätig ist. Wenn dies der Fall ist, wird sie lokal aus dem Cache ausgeliefert. Wenn sie nicht lokal verfügbar ist, wird unter der Voraussetzung, dass noch Downloadslots verfügbar sind, ein Download gestartet oder einem bereits bestehendem Download zur gleichen Access Unit beigetreten. Der Download wird als separater Thread gestartet (*HttpDownloader.writer*). Die Daten werden mit der Methode *reader* im *HttpDownloader* Objekt gelesen. Jede Access Unit wird in einer Datei gespeichert, die die Nummer der Access Unit als Namen hat. Schon während des Downloads werden die Daten in eine Datei gespeichert. Diese muss allerdings einen anderen Namen als die Nummer der Access Unit haben. Sobald der Download abgeschlossen ist, wird die Datei in die Access-Unit-Nummer umbenannt. Damit ist gewährleistet, dass atomar festgestellt werden kann, ob eine Access Unit lokal ausgeliefert werden kann oder nicht: Wenn eine Datei mit dem Nummer der Access Unit als Namen existiert, wurde die Access Unit vollständig heruntergeladen.

Durch die Verwendung temporärer Dateien wird der Speicherverbrauch des Proxies klein gehalten, da nur die Verwaltungsdaten im Arbeitsspeicher gehalten werden. Würde jeder Downloadthread noch die heruntergeladenen Daten speichern, würde dies relativ schnell zu erheblichem Bedarf an Arbeitsspeicher führen. Bei einer Access-Unit-Länge von 10 Sekunden und einer Datenrate von 1,6 Mbit/s ist eine Access Unit 2 MB groß. Bei zunehmender Länge der Access Units steigt der Speicherverbrauch entsprechend. Dasselbe gilt für höhere Bitraten bei der Codierung. Der verwendete Speicher kann zudem erst freigegeben werden, wenn alle Uploads, die ihn referenzieren, beendet wurden. Die sofortige Speicherung in Dateien bietet also erhebliche Vorteile.

Die Synchronisierung von Up- und Download geschieht über eine Liste von Semaphoren, die im Attribut *readersem* referenziert werden. Für jeden Upload existiert genau ein Semaphor. Dieser Semaphor kann beliebig oft erhöht werden und gibt an, wie oft die Methode *reader* den Systemruf *read* ausführen kann ohne auf das Ende der Datei zu stoßen. Die Anzahl der *read*-Aufrufe in der *writer*-Methode wird im Attribut *chunks* gespeichert. Wenn ein neuer Upload startet, wird der dazugehörige Semaphor auf

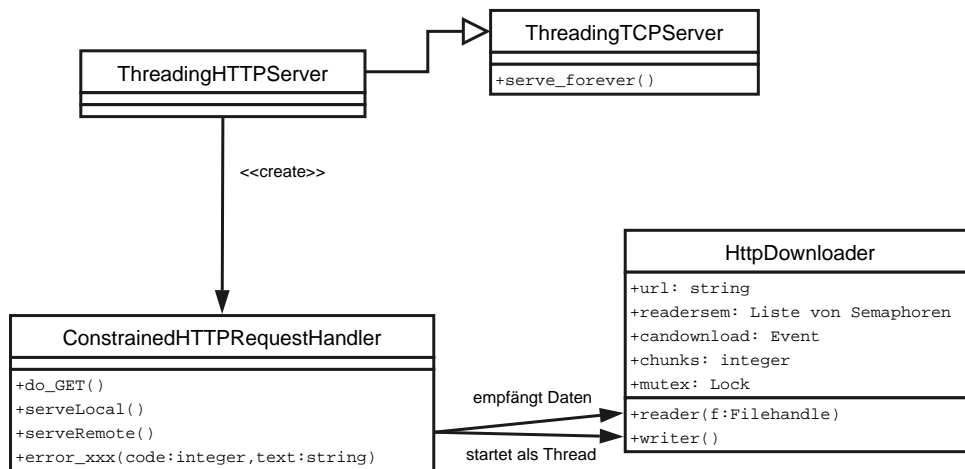


Abbildung 6.2: Zusammenspiel der Klassen im Proxy

diesen Wert initialisiert. Zugriffe auf das Attribut `chunks` und die Liste der Semaphore (`readersem`) werden über das Attribut `mutex` synchronisiert.

Daneben existieren noch globale Objekte, die wichtig für die Funktionalität sind und auf die ebenfalls exklusiver Zugriff gewährleistet ist: `uploadpool` und `downloadpool` sind Container-Objekte, die eine Maximalanzahl an enthaltenen Objekten gewährleisten. Darüber wird die Beschränkung der Bandbreite des Peers sichergestellt. Der Benutzer muss einstellen, wie viele Up- und Downloads er parallel gestatten will. Da davon ausgegangen wird, dass die Größe aller Access Units ähnlich ist (Ausnahme: Access Unit 0), kommt dies einer Bandbreitenregulierung gleich. Die Verwendung von TCP als Transportprotokoll trägt ihren Teil dazu bei.

Damit nicht mehr Verbindungen als nötig zum Tracker bestehen, existiert ein globales Objekt zur Verwaltung der bestehenden Verbindungen. In der Regel wird nur eine Verbindung existieren, es ist aber prinzipiell möglich, denselben Proxy für mehrere Videos zu verwenden. Für jede Ressource existiert dann genau eine Verbindung zum Tracker. Dies wird – entsprechend synchronisiert – durch das `trackers`-Objekt sichergestellt.

6.3 Mplayer-glue

`mplayer-glue` ist die Schnittstelle zwischen Proxy und Player. Als Videoplayer wurde MPlayer verwendet, weil MPlayer das verwendete Videoformat (Ogg Media) unterstützt, sehr robust funktioniert und sowohl für Linux/BSD als auch für Windows verfügbar ist (siehe auch [18]). MPlayer kann die Videodaten über die Standardeingabe lesen. `mplayer-glue` verwendet diese Funktion. Die Access Units werden vom Proxy über HTTP abgerufen und dann zusammengefügt auf die Standardeingabe des MPlayers ausgegeben. `mplayer-glue` hat eine primitive grafische Oberfläche, die notwendige Steuerfunktionen

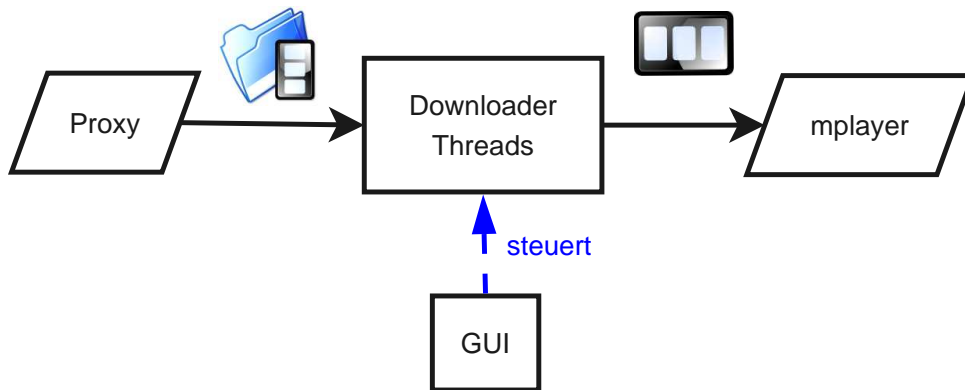


Abbildung 6.3: Struktur von mplayer-glue

bietet: Abspielen, Pause, Vor- und Zurückspulen sowie ans Ende springen. Der Sprung ans Ende wird für Liveübertragungen benötigt.

player-glue sorgt dafür, dass Access Units immer komplett an MPlayer übertragen werden. Bei Sprüngen werden dann entsprechend andere Access Units angefügt. Für die Verwendung in Netzen mit unterschiedlicher Up- und Downloadbandbreite kann mplayer-glue Access Units schon bevor sie abgespielt werden sollen anfordern. Die Pausenfunktion wird so realisiert, dass MPlayer keine Daten über die Standardeingabe mehr erhält. Alternativ zur Standardeingabe hätte MPlayer die Daten auch über HTTP abrufen können. Allerdings würde bei der Verwendung von HTTP eine Unterbrechung im Datenfluss zum Abbruch der Verbindung zwischen mplayer-glue und MPlayer führen. Eine Unterbrechung im Datenfluß würde zum Beispiel bei der Pausenfunktion auftreten. mplayer-glue würde dann einfach keine Daten über HTTP ausliefern, so dass MPlayer von einem Netzwerkproblem ausgehen würde. Die Datenübergabe über die Standardeingabe ist eine elegante Lösung für dieses Problem.

mplayer-glue besteht aus zwei Threads. Ein Thread ist für die Behandlung der grafischen Oberfläche zuständig, der andere für die Datenverarbeitung. Die Synchronisierung findet über ein Event-Objekt statt. Sofern Video gespielt werden soll, blockiert die *wait*-Funktion des Event-Objekts nicht.

6.4 C-Tracker

Die C-Implementierung des Trackers verwendet die in Kapitel 5.4 vorgestellten Datenstrukturen. Dadurch wird eine wesentlich höhere Leistung als bei der in Kapitel 6.1 vorgestellten Perl-Implementierung erreicht.

Der Tracker verwendet das Einprozessmodell für den Server. Allerdings wird nicht der *select*-Systemruf verwendet, sondern mit den *epoll*-Systemrufen von Linux 2.6 gearbeitet. Diese bieten eine wesentlich höhere Leistung und Skalierbarkeit. Die *epoll*-Systemrufe arbeiten laut [22] in $O(1)$. Außerdem erlauben sie eine deutlich größere Anzahl gleichzeitiger Verbindungen als *select*.

In der Implementierung wurden einige Einschränkungen im Protokoll vorgesehen: So ist die Zahl der möglichen Peers hart beschränkt, durch die Anpassung einer Konstanten in der Datei `peer.h` kann sowohl die Zahl der maximal möglichen Verbindungen als auch die maximale Zahl der Verbindungen zwischen den Peers eingestellt werden. Außerdem wurde die Anzahl der betrachteten Access Units zur Berechnung des Rankings der Peers und die maximale Anzahl Peers, die bei FINDPROXY zurückgeliefert wird, begrenzt. Dadurch kann eine konstante Laufzeit für die Funktion FINDPROXY erreicht werden.

Die Implementierung arbeitet mit begrenzten Puffern zur Eingabeverarbeitung. Dabei wird davon ausgegangen, dass pro read-Systemruf nur ein Befehl für den Tracker in den gelesenen Daten enthalten ist. Da Tracker und Peers synchron arbeiten, stellt dies keine Einschränkung der Funktionalität dar, vereinfacht aber die Implementierung. Der Befehlparger ist primitiv strukturiert. Er sucht die Befehlstrenner (meistens Leerzeichen) und ersetzt diese Trennzeichen durch das Nullzeichen (`\0`). Anschließend werden die Funktionen der C-Bibliothek verwendet, um diesen Text in das benötigte Datenformat (z.B. `struct in_addr`) zu konvertieren. Aus jedem Befehl wird eine Kommandodatenstruktur erstellt, die die benötigten Daten zur Verarbeitung enthält. Anschließend werden anhand der Kommandodatenstruktur die notwendigen Operationen auf der Peer-Datenbasis durchgeführt und die Ausgabe zur Übergabe an den Peer erzeugt.

Für jede Verbindung eines Peers existiert eine Datenstruktur. In dieser sind unter anderem die zu sendenden Daten enthalten. Bei der Befehlsverarbeitung werden die erzeugten Antwortdaten nicht sofort an den Peer gesendet, sondern zuerst in der Verbindungsdatenstruktur zwischengespeichert. Die dort eingetragenen Daten werden erst dann geschrieben, wenn `epoll` meldet, dass der Socket zum Peer schreibbar ist, um Blockieren beim Schreiben zu verhindern.

7 Evaluierung der Streaminglösungen

In diesem Kapitel sollen die vorgestellten Streaminglösungen miteinander verglichen werden. Die Kriterien sind das Verhalten beim Flash-Crowd-Syndrom, die maximale Anzahl an Teilnehmern, die versorgt werden können, und die Latenzzeit zwischen Anforderung der Daten und Start der Wiedergabe.

7.1 Streaming über HTTP

7.1.1 Versuchsaufbau

Die Versuche wurden auf dem Chemnitzer Linux Cluster (CLIC) [23] durchgeführt. Als Ausgangspunkt wurde *csn-lab3.csn.tu-chemnitz.de* verwendet. Dort war ein Apache 2 als Webserver installiert. Die Messungen wurden mit einem Python-Programm durchgeführt, das Daten per HTTP mit dem `urllib2`-Modul abholte und diese auf die Standardeingabe von `mplayer` ausgab. Das Python-Programm hat die Zeit zwischen Öffnen der Verbindung und Rückkehr der `urlopen`-Methode gemessen. Diese holt die HTTP-Kopfdaten vom Server ab, so dass der Request bereits bedient wird. Am Ende wurde gezählt, wie viele 4 kB-Blöcke abgeholt wurden. Jeder Testlauf lief 10 Minuten und wurde auf unterschiedlichen Knotenanzahlen durchgeführt. Während der Messungen war das Netz zwischen CLIC und *csn-lab3.csn.tu-chemnitz.de* in normaler Benutzung, bei Wiederholung können daher abweichende Ergebnisse erzielt werden.

Die Mediendaten lagen codiert auf *csn-lab3* vor, sie wurden mit dem Programm `lvstream` aus [18] erzeugt. Es wurde das hochauflösende Video verwendet. Die einzelnen Access Units wurden mit `cat` zusammengefügt und in einer Datei abgelegt.

7.1.2 Erwartetes Ergebnis

Es wird erwartet, dass das Video bis zur Sättigung der Leitung ohne Unterbrechung abgespielt werden kann. Das heißt, dass die Zahl der heruntergeladenen Blöcke nahe am erreichbaren Limit liegt. Die Sättigung der Leitung tritt bei 64 parallelen Downloads ein. Die erwarteten Latenzzeiten liegen deutlich unter einer Sekunde.

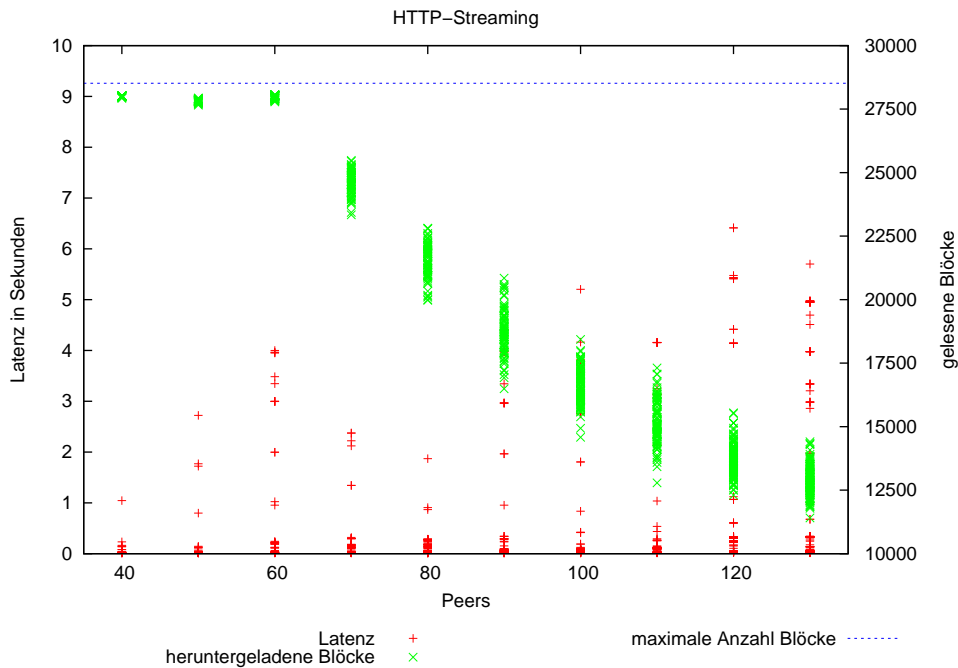


Abbildung 7.1: Latenzzeit und übertragene Blöcke beim HTTP Streaming

7.1.3 Ergebnisse

Die Messergebnisse zeigen eine Skalierung bis zum Limit der Netzverbindung. Sobald diese gesättigt war, sank die Zahl der übertragenen Datenblöcke, so dass die Wiedergabe stocken musste. Die Latenzzeit beim Zugriff steigt nicht wesentlich mit der Zahl der Teilnehmer. Es gab allerdings mit zunehmender Teilnehmerzahl eine größere Streuung bei den Latenzzeiten. Die Streuung der Zahl der heruntergeladenen Datenblöcke stieg nicht signifikant. In Abbildung 7.1 ist dies erkennbar.

Die Zahl der übertragenen Datenblöcke erreichte in keinem Fall das mögliche Limit. Dafür gibt es folgende Erklärung: `mplayer-feeder.py` stellt die Verbindung zum HTTP Server her, bevor `mplayer` gestartet wird. Der Grund dafür ist, dass die CPU-Zeit für das Starten von `mplayer` die Messergebnisse beim Öffnen verfälschen könnte. Die Zeit lief aber ab Öffnung der Verbindung, somit konnte die maximale Zahl nicht erreicht werden, weil die Initialisierungszeit von `mplayer` im Bereich von einigen Sekunden liegt. Da die Zahl der übertragenen Blöcke nicht wesentlich schwankt, kann davon ausgegangen werden, dass die Daten rechtzeitig beim Empfänger angekommen sind.

7.2 Streaming über RTSP/RTP

7.2.1 Versuchsaufbau

Die Versuche wurden im Chemnitzer StudentenNetz durchgeführt. Der Streamingserver wurde auf *csn-lab3.csn.tu-chemnitz.de* installiert. Die Clients verwendeten das Programm *openRTSP* aus der *liblive*-Bibliothek. Dieses empfängt den RTP-Datenstrom und stellt Statistiken über die empfangene Qualität auf, zählt also Paketverluste. Als Streamingserver wurde Apple Darwin Streaming Server Version 5.0.1.1 verwendet.

Ursprünglich sollte das Videomaterial inhaltlich mit dem im HTTP-Versuch verwendeten identisch sein. Es wurde lediglich in eine AVI-Datei umgewandelt, die mit Apple Quicktime in eine für Streaming geeignete MOV-Datei konvertiert wurde. Allerdings konnte *openRTSP* nicht mit den Daten umgehen und meldete Fehler. Daher wurde die beim Darwin Streaming Server mitgelieferte Beispieldatei *sample_100kbit.mp4* verwendet.

7.2.2 Erwartete Ergebnisse

Bis zur Sättigung der Leitung werden keine Paketverluste erwartet. Die Sättigung sollte bei knapp unter 1000 parallelen Streams eintreten.

7.2.3 Ergebnisse

In Abbildung 7.2 ist zu sehen, dass Apples Darwin Streaming Server es nicht schafft, die Fast-Ethernet-Leitung zu sättigen. Schon bei 350 Teilnehmern steigen die Paketverluste enorm an, so dass für viele Clients Probleme bei der Wiedergabe zu erwarten sind.

7.3 Peer-To-Peer-Netz mit Trackerimplementierung in Perl

7.3.1 Vorbetrachtungen

7.3.1.1 Latenzzeit

Für den Benutzer des Peer-To-Peer-Netzes ist die Gesamtlatenz interessant, also die Zeit, die es dauert, bis er sich die angeforderten Daten anschauen kann. Die Gesamtlatenzzeit teilt sich in Latenz, die durch das Peer-To-Peer-Netz verursacht wird, und andere Verzögerungen auf. Andere Verzögerungen sind die Zeit für die Kodierung und die Bereitstellung der Daten sowie die Durchlaufzeit für die Puffer im Player. Diese wurden in [18] untersucht und sind daher nicht Bestandteil dieser Arbeit. Die hier untersuchte Latenzzeit des Netzes setzt sich aus verschiedenen Komponenten zusammen:

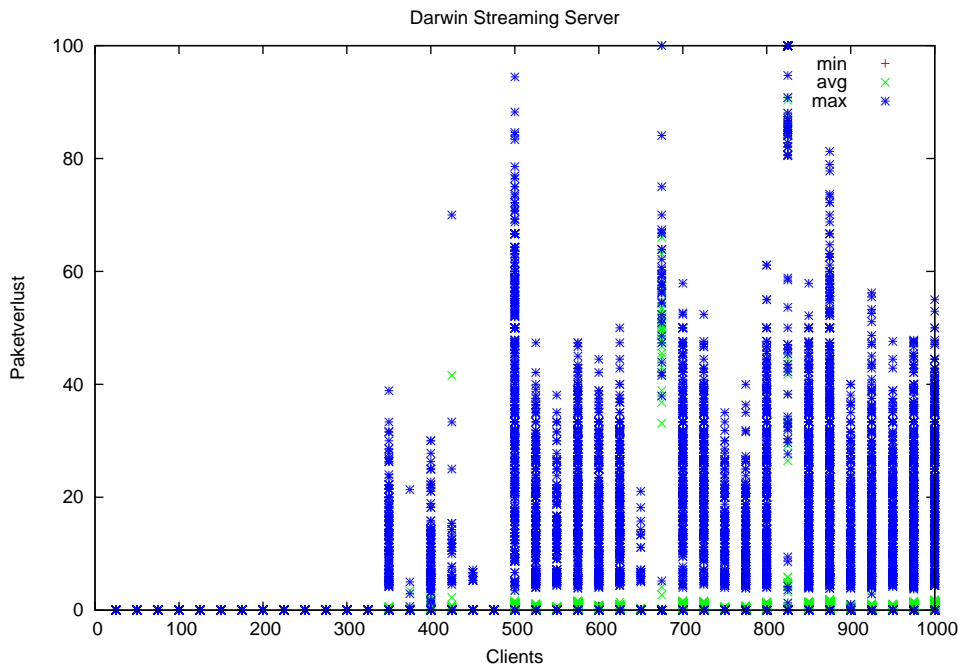


Abbildung 7.2: Entwicklung der Latenzzeiten bei Streaming über RTSP

- der Zeit, bis mplayer-glue Kontakt zum Proxy hergestellt hat (t_g)
- der Zeit, bis der Proxy Daten liefert ($t_p(x)$)
 - der Zugriffszeit für den Cache, wenn die Daten im Cache verfügbar sind (t_c)
 - der Zeit zum Beitreten eines bestehenden Downloads, falls bereits ein Download für die geforderte Access Unit existiert
 - Wenn ein Download gestartet werden muss:
 - * der Round-Trip-Time¹ zum Tracker (t_t). Diese wird nur benötigt, wenn kein übergeordneter Peer zur Verfügung steht
 - * dem Herstellen der Verbindung zu einem übergeordneten Proxy (t_v)
 - * der Zeit, bis der übergeordnete Proxy Daten liefert ($t_p(x-1)$)

¹Die Round-Trip-Time bezeichnet die Zeit, die zwischen dem Abschicken einer Anfrage und dem Empfangen der zugehörigen Antwort vergeht.

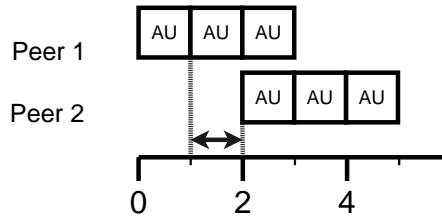


Abbildung 7.3: Cachedistanz

$$t = t_g + t_p(x) \quad (7.1)$$

$$t_p(0) = \begin{cases} t_c: & \text{wenn die Access Unit im Cache vorhanden ist} \\ t_j: & \text{wenn Download existiert} \end{cases} \quad (7.2)$$

$$\begin{aligned} t_p(x) &= t_{ix} + t_{vx} + t_p(x-1) \\ &= \sum_{i=1}^x (t_{ti} + t_{vi}) + \begin{cases} t_c: & \text{wenn die Access Unit im Cache vorhanden ist} \\ t_j: & \text{wenn Download existiert} \end{cases} \\ &= x(\bar{t}_t + \bar{t}_v) + \begin{cases} t_c: & \text{wenn die Access Unit im Cache vorhanden ist} \\ t_j: & \text{wenn Download existiert} \end{cases} \end{aligned} \quad (7.3)$$

$$t = t_g + x(\bar{t}_t + \bar{t}_v) + \begin{cases} t_c: & \text{wenn die Access Unit im Cache vorhanden ist} \\ t_j: & \text{wenn Download existiert} \end{cases} \quad (7.4)$$

Dabei gibt x die Pfadlänge des Requests an, also wie viele Peers traversiert werden. Aus der Formel kann man erkennen, dass die Latenzzeit von der Zahl der traversierten Peers abhängt.

7.3.1.2 Caching

Neben der Latenzzeit ist die Effektivität des Caches in jedem Proxy ein entscheidendes Kriterium für die Effektivität des Peer-To-Peer-Netzes. Mit der Cachedistanz wird der Abstand zwischen heruntergeladener Access Unit und der zu diesem Zeitpunkt letzten Access Unit, die der Proxy komplett heruntergeladen hat, bezeichnet. Damit wird also der zeitliche Versatz zwischen Daten lieferndem und empfangendem Peer gekennzeichnet (Siehe Abbildung 7.3). Über diesen kann dann die nötige Cachegröße bestimmt werden und eventuell geeignete Verdrängungsstrategien abgeleitet werden.

7.3.1.3 Datenratenabschätzung für den Tracker

Da sich der gesamte Overhead für die Verwaltung des Netzes beim Tracker summiert, ist eine Abschätzung für den zu erwartenden Traffic sinnvoll. Zunächst sollen die Datenmengen für die einzelnen Trackeroperationen aufgeschlüsselt werden. Dann soll noch der Overhead für Transport- und Vermittlungsschicht angegeben werden.

Operation	C⇒T in Bytes	C⇐T in Bytes
AVAILABLE	12 + <i>strlen</i> (\$BEGIN) + <i>strlen</i> (\$END)	7
DROP	7 + <i>strlen</i> (\$BEGIN) + <i>strlen</i> (\$END)	7
FINDPROXY mit Ranking	16 + <i>strlen</i> (\$URL) + <i>strlen</i> (\$IP_ADR) + <i>strlen</i> (\$PORT) + <i>strlen</i> (\$CLIENTS) + <i>strlen</i> (\$START_AU) + <i>strlen</i> (\$NR_PEERS)	13 + <i>strlen</i> (\$NR_PEERS) + \$NR_PEERS · (3 + <i>strlen</i> (\$IP_ADR) + <i>strlen</i> (\$PORT) + <i>strlen</i> (\$RANK))
FOUNDPROXY	14 + <i>strlen</i> (\$URL) + <i>strlen</i> (\$IP_ADR) + <i>strlen</i> (\$PORT)	7
DROPPROXY	13 + <i>strlen</i> (\$URL) + <i>strlen</i> (\$IP_ADR) + <i>strlen</i> (\$PORT)	7

Tabelle 7.1: Datenvolumen der einzelnen Trackeroperationen

In Tabelle 7.1 sind die Kosten der einzelnen Trackeroperationen aufgeschlüsselt. Dazu wird zur Vereinfachung mit folgenden Größen gerechnet: für die Textdarstellung der IP-Adressen werden 15 Bytes, für Ports 5 Bytes angenommen. Der Tracker gibt Rankings mit 5 Byte langen Zeichenketten aus – das entspricht drei Nachkommastellen. Access Units sollen 10 Sekunden Material enthalten. Die Zahl der Access Units soll mit dreistelligen Zahlen, also 3 Bytes, darstellbar sein, das reicht für eine Vorlesung. Die Länge der URL sei mit 256 Bytes angenommen. Das folgende Modell soll für die Berechnung der benötigten Übertragungsrate verwendet werden: die Peers verbinden sich in gleichmäßigem Abstand über einen vorgegebenen Zeitraum mit dem Tracker und spielen dann das Video ab. Jeder Peer fordert vom Tracker 20 Adressen an. Eine Adresse hat nach obigen Annahmen eine Länge von 28 Bytes.

Die übertragenen Daten setzen sich bei jedem Peer aus folgenden Bestandteilen zusammen: Einem Aufruf der Funktion FINDPROXY, einem Aufruf FOUNDPROXY und für jede übertragene Access Unit einem Aufruf von AVAILABLE. Die Anzahl der zu übertragenden Access Units berechnet sich wie folgt:

$$au(t_0) = \frac{t_I - t_0}{t_{AU}} \quad (7.5)$$

$$t_0 = \frac{i \cdot t_I}{n} \quad (7.6)$$

$$au(i) = \frac{t_I \left(1 - \frac{i}{n}\right)}{t_{AU}} \quad (7.7)$$

$$\begin{aligned} au_I(n, t_I, t_{AU}) &= \sum_{i=0}^{n-1} au(i) \\ &= \frac{t_I (n+1)}{2t_{AU}} \end{aligned} \quad (7.8)$$

Dabei ist t_I die Dauer des Intervalls, in dem alle Peers Kontakt zum Tracker aufnehmen, t_0 gibt den Zeitpunkt an, an dem ein Peer mit dem Abspielen beginnt. $au(t_0)$ gibt die Anzahl Access Units an, die ab Zeitpunkt t_0 noch im Intervall abspielbar sind. $au(i)$ gibt die Anzahl Access Units an, die Peer i abspielt. Dabei werden die Peers aufsteigend, nach dem Zeitpunkt, an dem sie mit der Wiedergabe begonnen haben, durchnummeriert. n ist die Zahl der Peers, die im Intervall mit der Wiedergabe beginnen. Die Gesamtzahl der im Intervall angeforderten Access Units wird schließlich durch au_I angegeben. Mit diesen Informationen kann schließlich die entstehende Daten- und Paketrate berechnet werden:

$$\begin{aligned} r_{fd}(n, t_I, t_{AU}) &= \frac{\overbrace{n(16 + 256 + 15 + 5 + 1 + 3 + 2)}^{\text{FINDPROXY}} + \overbrace{14 + 256 + 15 + 5}^{\text{FOUNDPROXY}}}{t_I} \\ &\quad + \frac{au_I(n, t_I, t_{AU}) \cdot \overbrace{(12 + 3 + 3)}^{\text{AVAILABLE}}}{t_I} \end{aligned} \quad (7.9)$$

$$r_{fu}(n, t_I, t_{AU}) = \frac{n(20 \cdot 28 + 13 + 2 + 7)}{t_I} + \frac{au_I(n, t_I, t_{AU}) \cdot 7}{t_I} \quad (7.10)$$

$$r_{sd}(n, t_{AU}) = 7 \frac{n}{t_{AU}} \quad (7.11)$$

$$r_{su}(n, t_{AU}) = 18 \frac{n}{t_{AU}} \quad (7.12)$$

$$p_{fd}(n, t_I, t_{AU}) = \overbrace{\frac{2n}{t_I}}^{\text{TCP-Handshake}} + \frac{2n}{t_I} + \frac{au_I(n, t_I, t_{AU})}{t_I} \quad (7.13)$$

$$p_{fu}(n, t_I, t_{AU}) = \overbrace{\frac{n}{t_I}}^{\text{TCP-Handshake}} + \frac{2n}{t_I} + \frac{au_I(n, t_I, t_{AU})}{t_I} \quad (7.14)$$

$$p_s(n, t_{AU}) = \frac{n}{t_{AU}} \quad (7.15)$$

Peers	Intervall	Flash-Crowd				Stabil		
		Bytes/s		Pakete/s		Bytes/s		Pakete/s
		C \Leftarrow T	C \Rightarrow T	C \Leftarrow T	C \Rightarrow T	C \Leftarrow T	C \Rightarrow T	
500	300 s	1145	1431	30	32	350	900	50
500	60 s	5025	5351	50	58	350	900	50
100000	300 s	229000	286000	6000	6333	70000	180000	10000
100000	60 s	1005001	1070001	10000	11667	70000	180000	10000

Tabelle 7.2: Nutzdatenraten am Tracker

Die Formeln 7.9 bis 7.12 geben die benötigte Datenrate an. Die Formeln 7.13ff geben die Anzahl der benötigten Pakete an. Dabei bezeichnet r Datenraten, p Paketraten, f Flashcrowd-Situationen, s stabile Situationen, u Upstream und d Downstream. Die Zahl der Pakete wird für die Berechnung des Overheads benötigt, da der Overhead pro Paket anfällt. Es wird davon ausgegangen, dass für jede Operation ein Paket Up- und Downstream notwendig ist. Lediglich beim TCP-Handshake fällt ein Paket mehr im Down- als im Upstream an. Jedes TCP-Paket hat einen Kopf von mindestens 20 Bytes Länge. Pro IP-Paket fallen in Version 4 ebenfalls mindestens 20 Bytes im Kopf an. Bei Version 6 sind sogar 40 Bytes Kopfdaten vorgesehen. Dabei ist zu beachten, dass in der Subnetzsicht möglicherweise Mindestgrößen für Pakete vorgesehen sind – bei Ethernet sind alle Pakete mindestens 64 Bytes groß. Außerdem sind die eventuell notwendigen ACK-Pakete für TCP nicht eingerechnet. Diese müssen gesendet werden, wenn die Abspieldauer der Access Units die maximale Zeit überschreitet, die TCP zum Bestätigen von Paketen zulässt.

In Tabelle 7.2 sind exemplarisch einige Fälle ausgerechnet. Dabei ist ersichtlich, dass selbst beim Eintreffen 100000 neuer Teilnehmer in einer Minute die Nutzdatenrate nur knapp über 1 MB/s liegt. Mit Overhead liegt die Datenrate etwa bei 1,5 MB/s. Der Overhead ist also nicht zu vernachlässigen. Trotzdem reicht ein normaler Fast Ethernet-Anschluß, wie er in Campusnetzen inzwischen typisch ist, für den Betrieb eines Trackers für so viele Teilnehmer aus.

7.3.2 Versuchsaufbau

Die Versuche wurden im Campusnetz der TU Chemnitz durchgeführt. Dabei liefen die Peer-Instanzen auf dem Chemnitzer Linux Cluster [23]. Es wurden jeweils 200 Knoten angefordert. Es wurde die Perl-Implementierung des Trackers für die Messungen verwendet und in 8 Instanzen mit Zugriff auf dieselbe Datenbank gestartet. Der Tracker lief auf *csn-lab4.csn.tu-chemnitz.de*, die Datenbank des Trackers ebenfalls auf *csn-lab4.csn.tu-chemnitz.de*. Die Quelle für den Videostrom war *csn-lab3.csn.tu-chemnitz.de*. Es ist zu beachten, dass die Messungen während des normalen Betriebes des Campusnetzes stattfanden und daher mit Abweichungen der Ergebnisse im Wiederholungsfall zu rechnen ist. Im realen Betrieb eines Peer-To-Peer-Netzes liegen allerdings auch keine Laborbedingungen vor, so dass die Messungen durchaus als Praxistest verstanden werden können.

In den Testreihen wurden verschiedene Parameter variiert:

Anzahl Peerinstanzen pro Knoten auf dem CLIC. Damit wurden in einem Vorversuch eventuelle Abhängigkeiten von der Belastung der einzelnen Knoten ermittelt.

Anzahl der Knoten zur Bestimmung der maximalen Anzahl Teilnehmer im Netz.

Anzahl der parallel erlaubten Downloads Damit sollte die Zahl der Peers ohne ausreichende Downloadkapazität verringert werden. Die Zahl der erlaubten Downloads ist in der Spalte *slots* in den Tabellen eingetragen.

Wiederverwendung von HTTP-Verbindungen Lohnt es sich, HTTP-Verbindungen wiederzuverwenden? (Connection: keep-alive) In den Tabellen bezeichnet der Wert *t* in der Spalte *keep* die Wiederverwendung von Verbindungen.

Maximale Wartezeit für Daten Dabei wurde der Parameter für die maximal zulässige Wartezeit für Downloads durch einen Zufallswert verändert. Der Bereich für die maximal zulässige Wartezeit lag bei 3 Sekunden oder bei 2,5-3,5 Sekunden. In den Tabellen bezeichnet der Wert *t* in der Spalte *time* die zufällige Variation.

Dabei wurde die Zeit zwischen Datenanforderung und -auslieferung gemessen. Wenn Peerlisten vom Tracker angefordert wurden, wurde die Round-Trip-Time gemessen. Die Cache-Treffer wurden ebenso wie die Pfadlänge gezählt.

Außer der Quelle konnte jeder Peer 5 andere Peers bedienen. Die Quelle bediente nur eine Verbindung zu einem anderen Peer.

Zur Messung wurde eine abgewandelte Version von *mplayer-glu*e verwendet, die ohne grafische Oberfläche auskommt und den *mplayer* mit den Optionen `-vo null -ao null -quiet` startet. Eine einfachere Möglichkeit wäre der schnellstmögliche Download der Daten gewesen. Allerdings hätte dies nicht den realen Bedingungen entsprochen, weil die Peers nicht die reale Abspielzeit benötigen würden. Außerdem würde die Wiedergabe nicht stocken, wenn die Daten nicht rechtzeitig vorhanden wären.

Im Vorfeld wurde eine Untersuchung durchgeführt, wie viele parallele Instanzen aus Proxy und Player ein Knoten des Clusters ausführen kann. Dabei wurde die Abspielzeit untersucht, da MPlayer die Daten nicht schnell genug abspielen kann, wenn die Prozessorleistung nicht ausreicht. Zwischen 1-3 Instanzen pro Knoten gab es keine relevanten Unterschiede bei der Abspielzeit. Darüber wurden die Abspielzeiten teilweise länger (4 Instanzen) bis deutlich zu lang (mehr als 4 Instanzen). Daher wurden die Messungen mit 3 Instanzen pro Knoten durchgeführt.

Von jeder Peer-Instanz wurde die Standardausgabe und die Fehlerausgabe von Proxy und *mplayer-glu*e abgespeichert. Dort sind die Messdaten verzeichnet. Diese wurden dann herausgefiltert, zusammengeführt und in eine PostgreSQL-Datenbank eingetragen. Mit dieser Datenbank wurden die Analysen durchgeführt.

7.3.3 Erwartetes Ergebnis

- Die durch das Peer-To-Peer-Netz eingeführte Latenzzeit liegt für den Großteil der Benutzer unter einer Sekunde.

Parameter			Anzahl Knoten. 3 Peers pro Knoten.							
slots	keep	time	25	50	75	100	125	150	175	200
1	t	t	0.977	0.954	0.944	0.950	0.954	0.954	0.966	0.976
1	t	f	0.954	0.962	0.961	0.962	0.960	0.939	0.966	0.967
1	f	t	0.977	0.971	0.968	0.961	0.950	0.959	0.959	0.956
2	t	t	0.977	0.956	0.965	0.951	0.953	0.957	0.956	0.967
3	t	t	0.963	0.958	0.955	0.945	0.956	0.963	0.970	0.934

Tabelle 7.3: Anteil der Downloads, die weniger als 0.5 Sekunden zum Starten benötigten

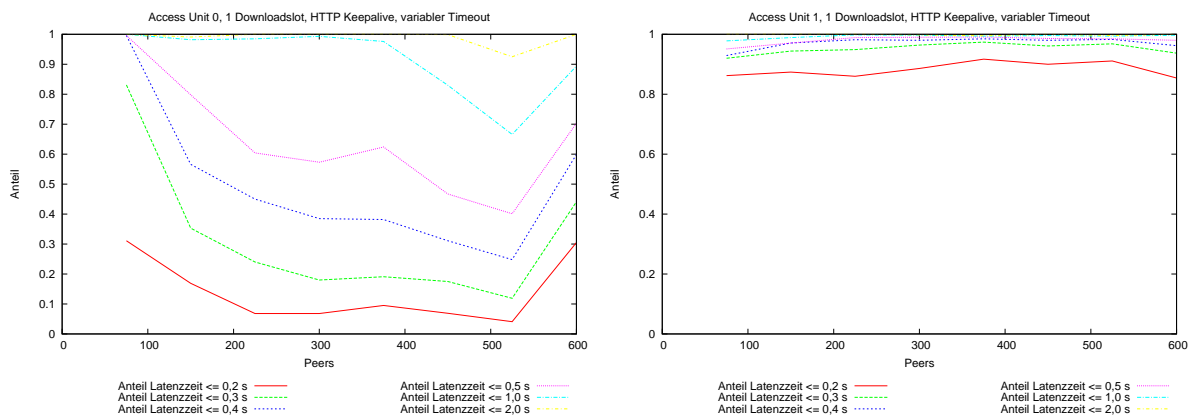


Abbildung 7.4: Anteil der Latenzzeiten für Access Units 0 und 1

- Die Latenzzeit für den Start von Downloads korreliert mit der Pfadlänge. Darin sei die Latenzzeit für Abfrage von Kontaktadressen nicht enthalten.
- Die Latenzzeit für Abfragen des Trackers hängt von der Zahl der Teilnehmer ab.
- Die Caches der Peers werden benutzt. Dabei ist die Cachedistanz bei allen Versuchen mit Flash-Crowd-Situation niedrig. Bei Versuchen ohne Flash-Crowd-Situation ist die Cachedistanz gleichmäßig verteilt.

7.3.4 Ergebnisse

Latenzzeit Bei der Analyse der Messdaten wurde zuerst der Anteil der Downloads, die schneller als 0,5 Sekunden starteten, errechnet. Die Ergebnisse sind in Tabelle 7.3 verzeichnet. Dabei ist ersichtlich, dass es bei den Ergebnissen im Rahmen der Messungenauigkeit keine Unterschiede in Abhängigkeit von den verwendeten Parametern gibt. Daher ist ein differenzierterer Blick auf die Ergebnisse notwendig. Die Verteilung der Latenzzeiten über die einzelnen Access Units zeigt, dass bei Access Unit 0 höhere Latenzzeiten zu verzeichnen sind als bei anderen Access Units. Das ist auch einleuchtend, da bei der ersten Access Unit zwingend eine Verbindung zum Tracker aufgebaut werden musste und eine Liste von möglichen Peers abgerufen wurde. Weiterhin kann man anhand der Tabellen 7.3, A.1 - A.6 sehen, dass der Anteil der Peers mit kurzen Latenzzeiten bei Access Unit 0 mit zunehmender Peeranzahl sinkt. Der

Knoten (3 Peers pro Knoten)	Latenzzeit in Sekunden
25	0.136
50	0.212
75	0.235
100	0.263
125	0.307
150	0.514
175	0.700
200	0.956

Tabelle 7.4: Durchschnittliche Latenzzeiten für FINDPROXY bei Access Unit 0

Grund ist in Tabelle 7.4 zu sehen: Die Zeit, bis FINDPROXY Ergebnisse liefert, steigt mit zunehmender Peeranzahl. Bei 200 Knoten (600 Peers) liegt die durchschnittliche Antwortzeit für FINDPROXY immerhin schon fast bei einer Sekunde. Der Tracker ist also ein kritischer Faktor für die Startlatenz der Peers. Die hohen Latenzzeiten sind der prototypischen Implementierung des Trackers in Perl und der Verwendung von PostgreSQL als Datenbank geschuldet. Mit der C-Implementierung des Trackers sind wesentlich bessere Ergebnisse zu erwarten.

Wie in Tabelle A.6 zu sehen ist, liegt der Anteil der Peers, die innerhalb von zwei Sekunden mit Abspielen beginnen konnten, selbst im schlechtesten Fall bei nicht weniger als 62,5%. Bei 525 Peers beträgt der Anteil der Peers bereits 74%. Allerdings erscheint die Verteilung der Anteile über die verschiedenen Versuche sehr schwankend, so dass eine mögliche Erklärung dafür in der Belastung des Netzes durch andere Nutzer zu suchen ist. Zwei Sekunden Zugriffszeit für Access Unit 0 sehen auf den ersten Blick zwar hoch aus, haben für den Nutzer allerdings kaum spürbare Auswirkungen, da mplayer-gluce Access Unit 0 sofort nach dem Start herunterlädt. Erst wenn der Nutzer die Wiedergabe startet, werden die weiteren Access Units heruntergeladen und abgespielt. Bei den weiteren Access Units gibt es aber, wie in den Tabellen A.1 bis A.6 zu sehen ist, für fast alle Nutzer nur Verzögerungen im Bereich von Sekundenbruchteilen. Solange das Peer-To-Peer-Netz die Daten schneller liefert als der Nutzer die Wiedergabe starten kann, wird er keine Verzögerung bemerken.

Korrelation zwischen Pfadlänge und Latenzzeit Zur Untersuchung der Korrelation zwischen Pfadlänge und Latenzzeit wurden die Daten in R [24] geladen und untersucht:

```
> library(RODBC)
> channel <- odbcConnect("PostgreSQL")
> dummy <- data.frame(sqlQuery(channel, "SELECT case when find_proxy
IS_NULL then latenz else latenz - find_proxy_end as latenz, pathlen
FROM datensatz WHERE latenz IS NOT NULL AND pathlen IS NOT NULL"))
> cor(dummy, use="pairwise.complete.obs")
      latenz pathlen
latenz 1.00000000 0.09862926
pathlen 0.09862926 1.00000000
```

Cache-Hits	Anzahl Knoten. 3 Peers pro Knoten.							
	25	50	75	100	125	150	175	200
1	0.583	0.583	0.579	0.544	0.533	0.531	0.516	0.543
2	0.245	0.247	0.244	0.254	0.255	0.264	0.273	0.255
3	0.116	0.114	0.111	0.121	0.130	0.124	0.123	0.115
4	0.039	0.040	0.046	0.054	0.055	0.054	0.058	0.056
5	0.013	0.014	0.013	0.022	0.023	0.023	0.023	0.024
6	0.002	0.002	0.006	0.004	0.003	0.003	0.005	0.006
7	0.000	0.000	0.001	0.001	0.000	0.001	0.001	0.000
8	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
9	0.000	0.000	0.001	0.000	0.000	0.000	0.000	0.000

Tabelle 7.5: Anteil der Cache-Treffer aufgeschlüsselt nach Peers

Das Ergebnis zeigt eine schwache Korrelation zwischen Pfadlänge und Latenzzeit. Das Ergebnis entspricht damit nicht den Erwartungen. Das bedeutet, dass die Pfadlänge keinen großen Einfluss auf die Latenzzeit bei der Übertragung hat. Der Einfluß der Pfadlänge auf die Latenzzeit wird also von anderen Faktoren überdeckt. Auch hier ist zu vermuten, dass einer dieser Faktoren die gleichzeitige Nutzung des Netzes durch andere Nutzer ist.

Cacheverhalten In Tabelle 7.5 sind die Anteile der Cache-Treffer nach Anzahl der Peers in den einzelnen Versuchen aufgeschlüsselt. Dabei ist ersichtlich, dass sich die Anteile der Cache-Treffer-Anzahlen abgesehen von Unterschieden im einstelligen Prozentbereich nicht ändert, wenn die Zahl der Teilnehmer im Peer-To-Peer-Netz verändert wird. Cache-Treffer wurden nur dann gezählt, wenn die angeforderte Access Unit vollständig im Cache vorlag. Beitritte zu bestehenden Downloads wurden *nicht* als Cache-Treffer gewertet.

In Abbildung 7.5 sind die Anteile der Cachedistanzen aufgezeichnet. Dabei ist ersichtlich, dass bei allen Versuchen mit Flash-Crowd-Situation der überwiegende Anteil der Cachedistanzen im Bereich zwischen 0 und 1 Access Unit liegt. Bei den zufällig verzögert gestarteten Peers verteilt sich die Cachedistanz über einen größeren Bereich.

7.3.5 Verdrängungsstrategie für den Cache

Die Versuche wurden mit unbegrenzter Cachegröße ausgeführt, um mögliche Seiteneffekte durch Verdrängungsstrategien auszuschließen. Von den klassischen Verdrängungsstrategien ist sicherlich nur FIFO für den Einsatz geeignet, weil LRU und LFU nur wirksam sind, wenn Zugriffe lokal sind, das heißt, dass Zugriffe auf dasselbe Datum zeitlich nah beieinander liegen. Dies ist aber beim Streaming nicht der Fall. Beim Streaming werden stattdessen aufeinander folgende Daten zeitlich nah beieinander abgerufen – FIFO ist also prädestiniert für diesen Anwendungsfall. Allerdings funktioniert FIFO nur, sofern die Cachegröße die Cachedistanz nicht unterschreitet. Sollte die Cachegröße die Cachedistanz unterschreiten, hätte der Cache keinerlei Effekt – die gespeicherten Access Units würden schon ersetzt werden, bevor

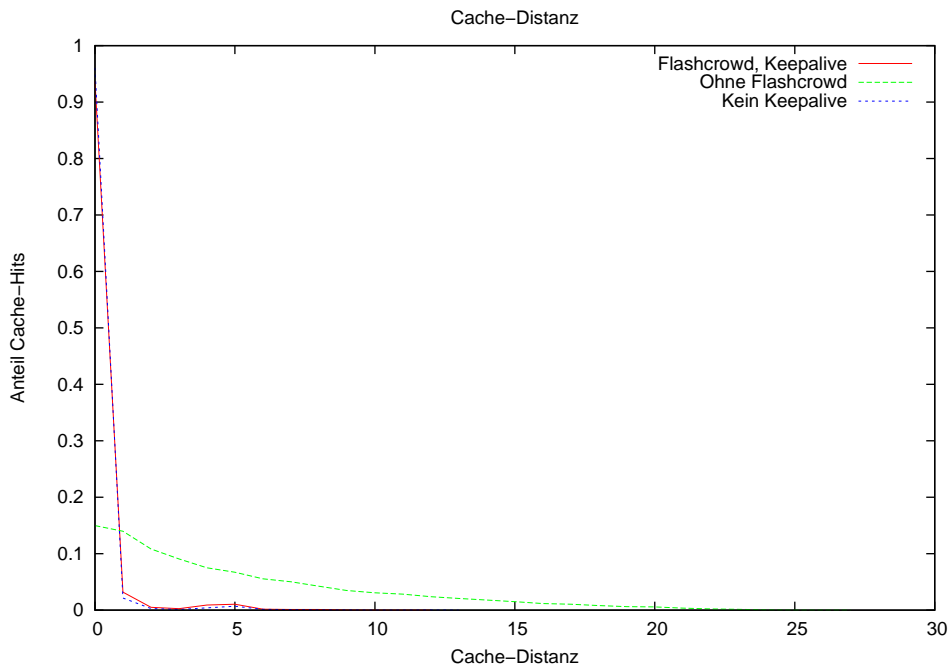


Abbildung 7.5: Cashedistanz

sie von einem anderen Peer angefordert worden wären. FIFO eignet sich besonders für Flash-Crowd-Situationen, da dann die Cashedistanzen gering sind.

Die in Kapitel 5.1 vorgeschlagene Berechnungsvorschrift für das Ranking sorgt dafür, dass der anfragende Peer anhand des Rankings ablesen kann, ob der vom Tracker vorgeschlagene Kandidat die gewünschte Access Unit in seinem Cache gespeichert hat oder nicht. Falls die Access Unit vorhanden ist, der Kandidat das Medium weiter abspielt und FIFO als Verdrängungsstrategie verwendet wird, kann der Peer daraus schließen, dass die Cachegröße des Kandidaten nicht kleiner als die Cashedistanz ist. Es handelt sich also um einen geeigneten Kandidaten. Allerdings kann es vorkommen, dass sich gar kein geeigneter Kandidat finden lässt. Dies könnte zum Beispiel der Fall sein, wenn nach einer Flash-Crowd-Situation ein einzelner Peer wesentlich später beitrifft und versucht, einen geeigneten Kandidaten zu finden. Wenn der Abstand zwischen diesem Peer und allen Kandidaten größer als deren Cachegröße ist, hat kein einziger Kandidat die geforderten Access Units vorrätig.

Eine alternative Verdrängungsstrategie sollte auch in einem solchen Fall gute Leistungen erbringen. Eine Strategie wäre das koordinierte Laden bestimmter Ausschnitte in den Cache, wenn Peers nicht Upstream anderer Peers sind, so dass für jede Access Unit immer mindestens ein Kandidat die Access Unit vorrätig hat. Dies kann auf verschiedene Arten organisiert werden (siehe Abbildung 7.6). Entweder verteilt man die Access Units in Gruppen (links) oder abwechselnd (rechts) auf mehrere Peers. Die Verteilung in Gruppen bietet sich an, wenn ein Peer als Quelle verwendet wird. Dann muss bei Verteilung auf n Peers $n - 1$ -mal die Quelle gewechselt werden. Abwechselnde Verteilung bietet sich an, wenn mehrere Quellen

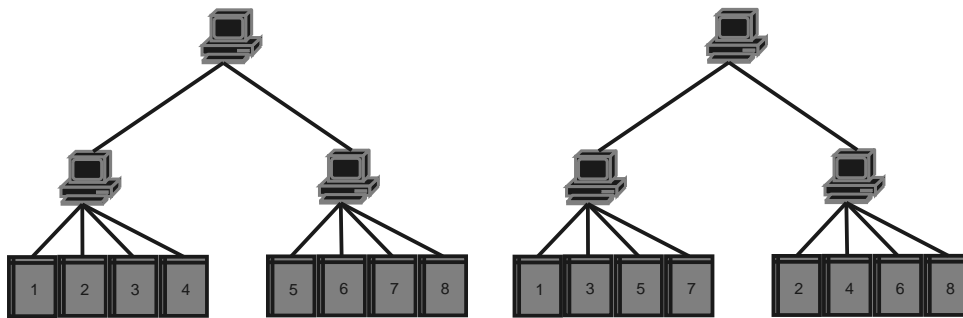


Abbildung 7.6: Verschiedene Methoden zur Clusterung der Access Units

verwendet werden, wie dies bei DSL-Nutzern der Fall sein wird. Dann werden von mehreren Quellen gleichzeitig Daten empfangen. Beide Verfahren können auch kombiniert werden.

7.4 Peer-To-Peer-Netz mit Trackerimplementierung in C

In Kapitel 7.3 konnte gezeigt werden, dass die prototypische Implementierung des Trackers in Perl bei 600 Teilnehmern ihre Leistungsgrenze erreicht. Daher wurde ein weiterer Prototyp implementiert – diesmal wurden die in Kapitel 5.4 beschriebenen Datenstrukturen und C statt Perl als Programmiersprache verwendet.

Der in C implementierte Prototyp läßt eine weitaus höhere Skalierbarkeit erwarten als die Perl-Implementierung. Daher mussten entsprechende Vorkehrungen getroffen werden um wesentlich mehr Peers zu simulieren. So war es mit den zur Verfügung stehenden Ressourcen unmöglich, 100000 Peers zu simulieren. Daher wurde die Simulationsumgebung abgeändert. Die Peers rufen voneinander keine Daten mehr ab, sondern verbinden sich nur noch mit dem Tracker und verhalten sich gegenüber dem Tracker wie normale Peers. Zuerst wurde dies mit einem einfachen Perl-Programm durchgeführt, das mehrfach parallel gestartet wurde. Doch auch damit sind nur kleine Zahlen an Peers simulierbar, da sich der Speicherverbrauch pro Prozess schnell zu relevanten Größen summiert. Also wurde eine C-Implementierung des Peersimulators erstellt. Diese wurde auf einem Computerserver gestartet. Dabei stellte sich heraus, dass die Zahl der pro Nutzer erlaubten Prozesse eine Beschränkung darstellt. Auf dem leistungsfähigsten Computerserver der Universität konnten damit aber immerhin 24000 Peers simuliert werden. Damit wurde die CPU des Trackers aber nur zu etwa 5% ausgelastet. Um in noch mehr Peers simulieren zu können, wurde das Programm auf dem CLIC mehrfach auf verschiedenen Knoten gestartet. Damit konnten bis zu 50000 Peers simuliert werden. Die Begrenzung stellte hier die Leistungsfähigkeit der einzelnen Knoten auf dem CLIC und die Verteilung der Messdaten dar. Die Daten konnten nicht mit MPI verteilt werden, da die MPI-Implementierung es nicht gestattet, in MPI-Programmen den fork-Systemruf auszuführen. Daher wurde ein neues Programm erstellt, was mittels select-Systemruf mehrere Peers in einem Prozess simuliert und die Messergebnisse mit MPI sammelt. Damit konnten 100000 Peers simuliert werden.

7.4.1 Versuchsaufbau

Die Versuche wurden auf dem CLIC durchgeführt. Der Tracker lief auf der Hardware von *csn-lab4.csn.tu-chemnitz.de*. Allerdings befindet sich zwischen CSN-Labor und CLIC der *csn-server*. Dort ist eine stateful Firewall aktiv. Da diese bei 100000 Verbindungen zur Verfälschung von Messergebnissen führen könnte, wurde die Hardware von *csn-lab4.csn.tu-chemnitz.de* während der Versuche vor *csn-server* platziert, so dass Verfälschungen durch stateful Firewalls ausgeschlossen sind. Im Campusnetz der TU Chemnitz befinden sich laut Aussage des Rechenzentrums keine stateful Firewalls zwischen CSN und CLIC (siehe Abschnitt C.2).

Auf *csn-lab4.csn.tu-chemnitz.de* wurde die Zahl der maximal im System verfügbaren Filedeskriptoren auf 102285 eingestellt. Die Einstellungen in `/etc/security/limits.conf` wurde entsprechend angepasst. Der Tracker wurde mit gcc Version 3.3.5 aus Debian Sarge mit den Optionen `-g -Wall -pedantic -std=c99 -Wpointer-arith -Wshadow -Wwrite-strings -Wstrict-prototypes -Wmissing-prototypes -Wcast-qual -Waggregate-return -Wmissing-declarations -Wnested-externs -Winline -Werror -W` übersetzt. Mit den gleichen Optionen, aber gcc-Version 2.96 20000731 aus Redhat Linux 7.3, wurde der Trackerclient übersetzt.

7.4.2 Erwartetes Ergebnis

Der Tracker ist in der Lage, bis zu 100000 Verbindungen gleichzeitig zu bedienen. Die erwartete Latenzzeit liegt bei allen Operationen im Durchschnitt deutlich unter einer Sekunde.

7.4.3 Ergebnis

In Abbildung 7.7 sind die durchschnittlichen Latenzzeiten der verwendeten Operationen abgebildet. Wie erwartet liegen die Latenzzeiten deutlich unter einer Sekunde, auch bei 100000 parallelen Verbindungen. Die deutlich höheren Latenzzeiten für FINDPROXY und FOUNDPROXY gegenüber AVAILABLE erklären sich aus der Testkonstellation. FINDPROXY und FOUNDPROXY wurden überwiegend während einer Flashcrowdsituation aufgerufen, während große Teile der AVAILABLE-Befehle während einer stabilen Situation aufgerufen wurden. Durch die deutlich höhere Zahl der AVAILABLE-Aufrufe ist der Anteil der Meßergebnisse unter Flash-Crowd-Einfluß geringer. Damit sinkt auch die durchschnittliche Latenzzeit.

Der Einfluß der Flashcrowdsituation läßt sich sehr gut an den Meßergebnissen der FOUNDPROXY-Aufrufe ablesen. Obwohl die FOUNDPROXY-Implementierung in $O(1)$ abläuft, nähert sich Kurve der Latenzzeit an eine Gerade mit positivem Anstieg an. Dabei sagt die Kostenabschätzung von $O(1)$ nur aus, dass die Ausführungszeit einer *einzelnen* Operation nicht von der Zahl der Verbindungen abhängt. Bei einer entsprechend steigenden Zahl Befehle steigt auch die durchschnittliche Latenzzeit, weil die Wahrscheinlichkeit sinkt, dass der Befehl sofort bedient werden kann. An der Kurve von FINDPROXY kann man den Mehraufwand gegenüber FOUNDPROXY sehen - FINDPROXY führt deutlich mehr Operationen durch.

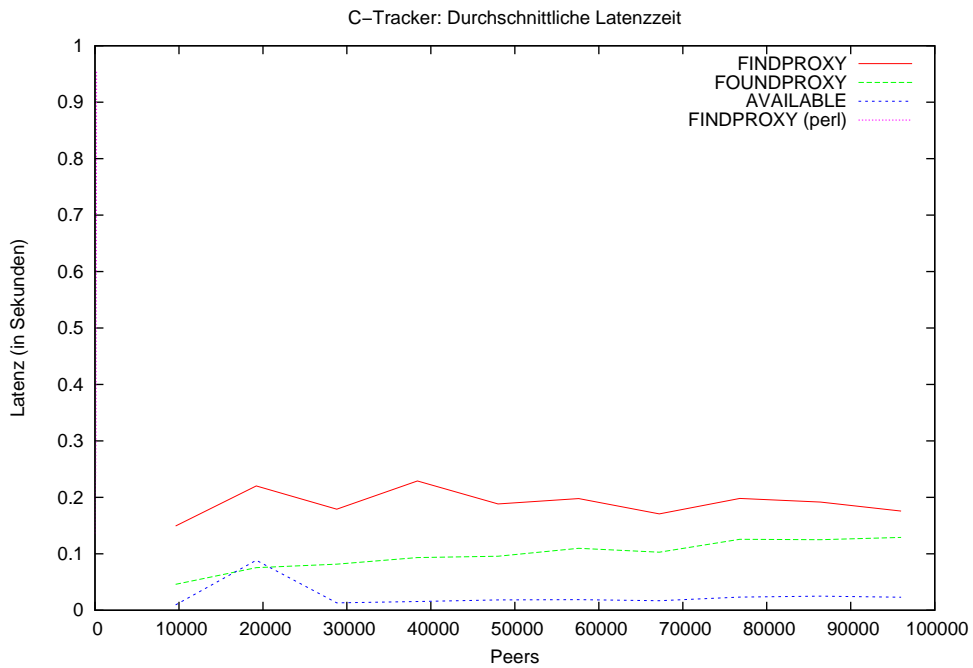


Abbildung 7.7: Durchschnittliche Latenzzeit der einzelnen Operationen für die C-Implementierung des Trackers

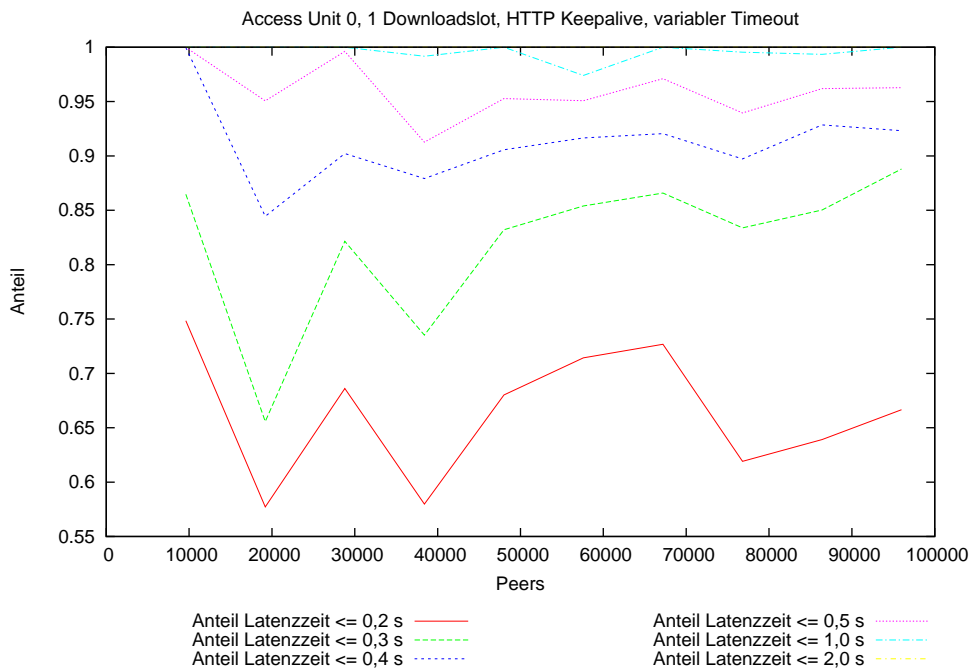


Abbildung 7.8: Anteil der Latenzzeiten bei FINDPROXY

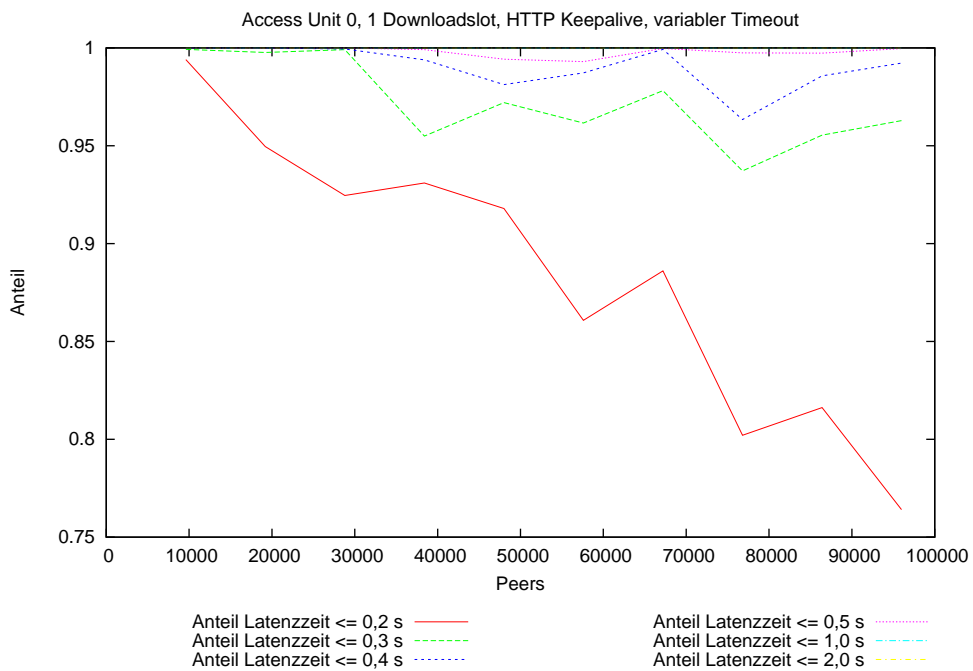


Abbildung 7.9: Anteil der Latenzzeiten bei FOUNDPROXY

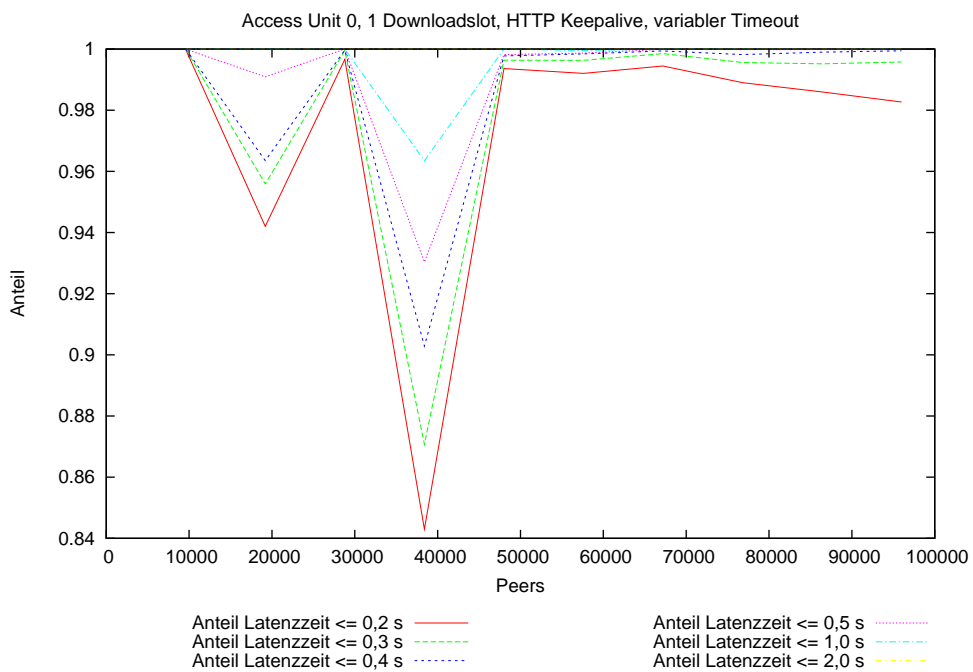


Abbildung 7.10: Anteil der Latenzzeiten bei AVAILABLE

Abbildungen 7.8 bis 7.10 stellen die Anteile der Latenzzeiten bei den einzelnen Operationen dar. Bei 19200 und 38400 simulierten Peers scheint es Aussetzer gegeben zu haben, die dafür gesorgt haben, dass die Ergebnisse aus dem üblichen Rahmen fallen. Hierzu sei noch einmal auf die parallele Benutzung sowohl des Campusnetzes als auch des CSN verwiesen. Insgesamt entspricht das Ergebnis damit den Erwartungen.

8 Zusammenfassung

In dieser Arbeit wurden verschiedene Möglichkeiten des Streaming und unterschiedliche Ansätze zur Konstruktion von Peer-To-Peer-Netzen vorgestellt. Daraus wurde ein Konzept für ein Peer-To-Peer-Netz erstellt, mit dem sowohl Live- als auch On-Demand-Videoströme übertragen werden können. In Versuchen an einem Prototyp wurden insbesondere Latenzzeiten begutachtet. Dabei wurde festgestellt, dass die erzielten Latenzzeiten für den in Kapitel 2 beschriebenen Einsatzzweck ausreichend gering sind. Gegenüber den etablierten Verteilungstechnologien konnte im Versuch eine deutliche Steigerung der Zahl gleichzeitiger Teilnehmer nachgewiesen werden.

A Tabellierte Messergebnisse

Parameter			Anzahl Knoten. 3 Peers pro Knoten.							
slots	keep	time	25	50	75	100	125	150	175	200
Access Unit 0										
1	t	t	0.311	0.169	0.068	0.068	0.095	0.069	0.041	0.305
1	t	f	0.231	0.100	0.031	0.059	0.032	0.039	0.025	0.094
1	f	t	0.187	0.127	0.047	0.074	0.034	0.038	0.053	0.029
2	t	t	0.111	0.067	0.077	0.076	0.038	0.069	0.043	0.127
3	t	t	0.140	0.170	0.098	0.060	0.059	0.038	0.025	0.102
Access Unit 1										
1	t	t	0.862	0.874	0.860	0.886	0.917	0.900	0.911	0.854
1	t	f	0.827	0.853	0.844	0.907	0.887	0.867	0.900	0.871
1	f	t	0.858	0.858	0.879	0.897	0.876	0.887	0.915	0.895
2	t	t	0.898	0.881	0.873	0.896	0.900	0.918	0.891	0.910
3	t	t	0.793	0.803	0.859	0.863	0.863	0.894	0.914	0.905

Tabelle A.1: Anteil der Downloads, die weniger als 0.2 Sekunden zum Starten benötigten

Parameter			Anzahl Knoten. 3 Peers pro Knoten.							
slots	keep	time	25	50	75	100	125	150	175	200
Access Unit 0										
1	t	t	0.831	0.353	0.240	0.180	0.191	0.175	0.119	0.441
1	t	f	0.578	0.351	0.145	0.159	0.079	0.084	0.068	0.122
1	f	t	0.596	0.324	0.193	0.188	0.090	0.099	0.100	0.044
2	t	t	0.427	0.213	0.216	0.206	0.120	0.120	0.116	0.146
3	t	t	0.687	0.388	0.219	0.165	0.118	0.086	0.073	0.170
Access Unit 1										
1	t	t	0.920	0.944	0.949	0.964	0.974	0.961	0.968	0.937
1	t	f	0.938	0.933	0.919	0.963	0.966	0.934	0.962	0.943
1	f	t	0.924	0.911	0.945	0.959	0.940	0.959	0.969	0.974
2	t	t	0.960	0.955	0.950	0.962	0.971	0.975	0.956	0.970
3	t	t	0.880	0.875	0.935	0.938	0.948	0.962	0.967	0.975

Tabelle A.2: Anteil der Downloads, die weniger als 0.3 Sekunden zum Starten benötigten

Parameter			Anzahl Knoten. 3 Peers pro Knoten.							
slots	keep	time	25	50	75	100	125	150	175	200
Access Unit 0										
1	t	t	0.996	0.566	0.450	0.385	0.382	0.311	0.248	0.599
1	t	f	0.764	0.531	0.443	0.324	0.230	0.157	0.135	0.154
1	f	t	0.982	0.676	0.433	0.360	0.191	0.199	0.160	0.059
2	t	t	0.707	0.409	0.446	0.386	0.236	0.186	0.181	0.171
3	t	t	1.000	0.637	0.389	0.451	0.297	0.132	0.140	0.237
Access Unit 1										
1	t	t	0.929	0.971	0.982	0.980	0.985	0.980	0.983	0.962
1	t	f	0.956	0.951	0.947	0.982	0.986	0.956	0.973	0.973
1	f	t	0.938	0.929	0.959	0.982	0.964	0.982	0.985	0.984
2	t	t	0.978	0.970	0.980	0.980	0.989	0.993	0.969	0.990
3	t	t	0.900	0.893	0.957	0.957	0.968	0.987	0.982	0.988

Tabelle A.3: Anteil der Downloads, die weniger als 0.4 Sekunden zum Starten benötigten

Parameter			Anzahl Knoten. 3 Peers pro Knoten.							
slots	keep	time	25	50	75	100	125	150	175	200
Access Unit 0										
1	t	t	0.996	0.798	0.604	0.573	0.624	0.467	0.401	0.703
1	t	f	0.862	0.720	0.759	0.513	0.417	0.293	0.205	0.182
1	f	t	0.996	0.949	0.684	0.549	0.334	0.326	0.238	0.087
2	t	t	0.933	0.638	0.677	0.615	0.372	0.262	0.287	0.199
3	t	t	1.000	0.792	0.553	0.725	0.455	0.177	0.219	0.281
Access Unit 1										
1	t	t	0.951	0.971	0.989	0.990	0.991	0.986	0.985	0.980
1	t	f	0.960	0.956	0.956	0.993	0.990	0.968	0.982	0.984
1	f	t	0.956	0.940	0.964	0.987	0.976	0.989	0.992	0.988
2	t	t	0.982	0.978	0.992	0.990	0.992	0.995	0.978	0.996
3	t	t	0.913	0.945	0.971	0.962	0.977	0.993	0.987	0.994

Tabelle A.4: Anteil der Downloads, die weniger als 0.5 Sekunden zum Starten benötigten

Parameter			Anzahl Knoten. 3 Peers pro Knoten.							
slots	keep	time	25	50	75	100	125	150	175	200
Access Unit 0										
1	t	t	1.000	0.982	0.985	0.993	0.976	0.830	0.666	0.894
1	t	f	0.996	0.822	0.991	0.948	0.956	0.651	0.507	0.369
1	f	t	1.000	1.000	0.993	0.906	0.949	0.907	0.674	0.248
2	t	t	1.000	0.980	0.990	0.988	0.866	0.650	0.711	0.373
3	t	t	1.000	1.000	0.975	0.988	0.977	0.560	0.475	0.542
Access Unit 1										
1	t	t	0.978	0.989	0.998	0.997	0.994	0.996	0.994	0.997
1	t	f	0.991	0.989	0.982	0.999	0.997	0.984	0.995	0.999
1	f	t	0.978	0.971	0.982	0.994	0.996	0.999	1.000	0.998
2	t	t	1.000	1.000	1.000	0.998	0.995	0.998	0.994	0.999
3	t	t	0.953	0.979	0.991	0.973	0.993	0.999	0.991	0.998

Tabelle A.5: Anteil der Downloads, die weniger als 1.0 Sekunden zum Starten benötigten

Parameter			Anzahl Knoten. 3 Peers pro Knoten.							
slots	keep	time	25	50	75	100	125	150	175	200
Access Unit 0										
1	t	t	1.000	0.991	1.000	1.000	1.000	0.999	0.925	1.000
1	t	f	1.000	0.987	1.000	1.000	0.999	0.878	0.746	0.761
1	f	t	1.000	1.000	0.997	0.999	1.000	1.000	0.990	0.625
2	t	t	1.000	1.000	1.000	1.000	0.999	0.957	0.976	0.769
3	t	t	1.000	1.000	1.000	1.000	1.000	0.908	0.831	0.918
Access Unit 1										
1	t	t	1.000	1.000	1.000	0.998	0.995	1.000	0.996	1.000
1	t	f	1.000	0.998	1.000	1.000	0.999	0.995	0.997	0.999
1	f	t	1.000	1.000	0.996	0.999	1.000	0.999	1.000	1.000
2	t	t	1.000	1.000	1.000	1.000	0.999	0.999	1.000	1.000
3	t	t	1.000	1.000	1.000	1.000	0.999	1.000	1.000	1.000

Tabelle A.6: Anteil der Downloads, die weniger als 2.0 Sekunden zum Starten benötigten

B Befehlsreferenz

B.1 Tracker

Der Tracker wird mit dem Befehl `trackerd.pl` gestartet. Die Form der Kommandozeile folgt den bei freien Unix-Programmen üblichen Konventionen zur Parameterübergabe. Zusätzlich zu den in `Net::Server` definierten Parametern sind folgende Parameter möglich:

<code>--db-name</code>	Name der zu verwendenden Datenbank
<code>--db-host</code>	Name des Datenbankservers
<code>--db-user</code>	Der zu verwendende Datenbanknutzer
<code>--db-password</code>	Das Datenbankpasswort

B.2 Proxy

Der Proxy wird mit dem Befehl `threadinghttpd.py -t tracker:port` gestartet. Die Parameter werden mit `getopt` eingelesen, so dass die bei freien Unix-Programmen üblichen Konventionen zur Parameterübergabe gelten.

<code>-h, --help</code>	Zeigt eine Parameterübersicht an
<code>-c, --cache-dir=CACHE</code>	Das Verzeichnis, in dem der Proxy die Cache-Einträge abspeichert. Standard: <code>cache</code>
<code>-d, --max-downloads=DOWNLOADS</code>	Maximale Zahl paralleler Downloads. Standard: 2
<code>-u, --max-uploads=UPLOADS</code>	Maximale Zahl paralleler Uploads. Standard: 5
<code>-p, --port=PORT</code>	Der Proxy nimmt Verbindungen auf Port <code>PORT</code> an. Standard: 8080
<code>-t, --tracker=TRACKER</code>	Der zu verwendende Tracker. <code>TRACKER</code> hat das Format <code>Rechnername:Port</code> . Dieser Parameter ist zwingend notwendig.
<code>-T, --timeout=TIMEOUT</code>	Gibt die Zeit an, die bei Downloads auf Daten gewartet wird.
<code>-k, --no-keep-alive</code>	setzt den HTTP-Kopf <code>Connection</code> auf <code>close</code> . Standard: <code>Connection: keep-alive</code>

B.3 Mplayer-glue

Mplayer-glue wird mit `mplayer-glue.py` gestartet. Als einzigen Parameter bekommt es die Basis-URL der abzuspielenden Ressource übergeben. Die Adresse des Proxies wird in der Umgebungsvariable `http_proxy` angegeben.

C Hardwareumgebung

C.1 Rechnerspezifikation

	csn-lab3	csn-lab4	Knoten auf dem CLIC
CPU	AMD Athlon64 3000+	Intel Celeron 2.4 GHz	Intel Pentium 3 800 MHz
RAM	1 GB	1 GB	512 MB
Netzwerkschnittstelle	Realtek 8169	Realtek 8139C	2 x Level One FNC 0108TX
Betriebssystem	SuSE Linux 9.2 64 Bit	Debian Linux 3.1 (Prerelease)	Redhat Linux 7.3
Software	Apache 2.0.52	Perl 5.8.4, PostgreS-QL 7.4.7	Python 2.4, MPlayer 1.0pre6-3.1

C.2 Netz

	Typ	Netzebene	Netzschnittstellen
mpr-54-r	Cisco 6506 mit Supervisor Engine 2	3	48x1000T, 6x1000SX, 3x1000LX
mpr-72-k	3com 3300FX	2	8x100FX, 2x100TX, 1x1000LX
mpr-72-0	3com 3300 Classeic (3C16980A)	2	24x100TX, 1x100FX
csn-server	Linux PC (Dual Intel Xeon 2.4 GHz, 2 GB RAM, Kernel 2.4.30)	3	2x1000T (Intel E1000)

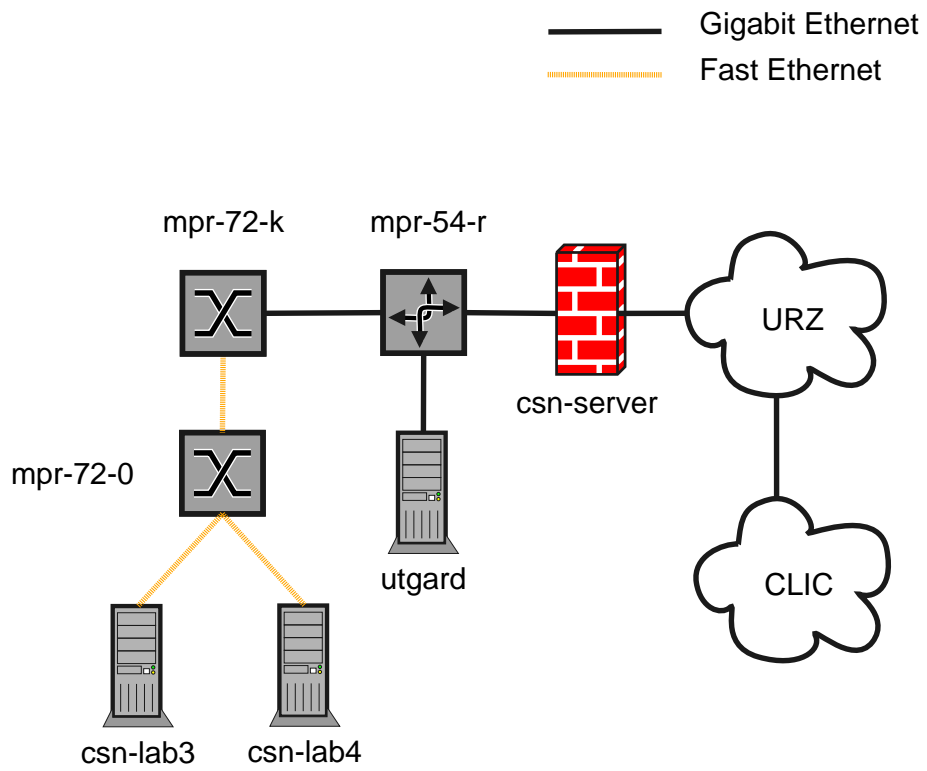


Abbildung C.1: Netzaufbau

D Glossar

ARP Adress Resolution Protocol. Dient zur Abbildung von IP-Adressen auf Adressen der untergeordneten Netzwerkschicht.

CPAN Comprehensive Perl Archive Network. Sammlung von Perl-Modulen.

DHT Distributed Hash Table. Eine Hashtabelle, die über mehrere Rechner verteilt gespeichert wird.

DNS Domain Name System. System zur Abbildung von Namen auf Adressen im Internet.

DSL Digital Subscriber Line. Verbreitete Zugangstechnologie zum Internet.

Flash-Crowd-Syndrom Eintreffen neuer Netzteilnehmer in sehr kurzer Zeit.

HTL Hops To Live. Ein Zähler, der angibt, wie viele Knoten eine Nachricht traversieren darf.

HTTP Hypertext Transport Protocol. Verbreitetes Dateiübertragungsprotokoll im Internet.

IP Internet Protocol.

IRC Internet Relay Chat. Gruppenkommunikationssystem auf Textbasis im Internet.

ISDN Integrated Services Digital Network.

ISP Internet Service Provider. Anbieter eines Internetzugangs.

MAC Media Access Control.

MAT MAC Adress Translation. Systematische Veränderung der MAC-Adresse von Netzwerkpaketen.

MPEG Moving Picture Experts Group. Standardisierungsgremium für Videokompression.

Multicast Senden eines IP-Pakets an mehrere Empfänger. Das Paket wird auf den Routern entsprechend repliziert.

NAT Network Adress Translation. Systematische Veränderung der IP-Adresse von IP-Paketen.

Peer Teilnehmer in einem Peer-To-Peer-Netz.

QoS Quality of Service. Netzwerkpakete werden anhand gegebener Regeln gegenüber anderen priorisiert.

RFC Request for Comment. Internetstandard.

RDP Reliable Datagram Protocol. Wenig verwendetes Transportprotokoll im Internet.

Round-Trip-Time Die Round-Trip-Time bezeichnet die Zeit, die zwischen dem Abschicken einer Anfrage und dem Empfangen der zugehörigen Antwort vergeht.

RPC Remote Procedure Call. Über eine Netzverbindung wird eine Funktion aufgerufen.

RSVP Resource Reservation Protocol.

RTCP RTP Control Protocol. Überträgt Empfangsstatistiken für RTP-Ströme.

RTP Real Time Transport Protocol. Wird zur Übertragung von Mediendaten im Internet verwendet.

RTSP Real Time Streaming Protocol. Wird zur Steuerung von RTP-Strömen verwendet.

Servent Anderer Begriff für Peer.

TCP Transmission Control Protocol. Verbreitetes Transportprotokoll im Internet für Verbindungsbasierte Kommunikation.

Timeout Überschreitung eines vorgegebenen Zeitlimits für eine Operation.

UDP User Datagram Protocol. Verbreitetes Transportprotokoll im Internet für verbindungslose Kommunikation.

URL Uniform Resource Locator. Addressierungsschema für Ressourcen im Internet.

TTL Time To Live. Zähler, der angibt wieviele Router ein Paket noch traversieren darf.

Literaturverzeichnis

- [1] R. Fielding, J. Gettys, J. Mogul, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1, 1999.
- [2] Transmission control protocol, 1981.
- [3] H. Schulzrinne, S. Casner, R. Frederick, and v. Jacobson. Rtp: A transport protocol for real-time applications, 1996.
- [4] L. Zhang, S. Berson, S. Herzog, and s. Jamin. Resource reservation protocol (rsvp) - version 1 functional specification, 1997.
- [5] H. Schulzrinne, a. Rao, and R. Lanphier. Real time streaming protocol (rtsp), 1998.
- [6] User datagram protocol, 1980.
- [7] Version 2 of the reliable data protocol (rdp), 1981.
- [8] Ethendranath Bommaiah, Katherine Guo, Markus Hofmann, and Sanjoy Paul. Design and implementation of a caching system for streaming media over the internet.
- [9] Andy Oram, editor. *Peer-To-Peer*. O'Reilly, 2001.
- [10] drscholl@users.sourceforge.net. Napster messages. <http://opennap.sourceforge.net/napster.txt>, 2000.
- [11] Bram Cohen. Incentives build in bittorrent. <http://bittorrent.com/bittorrentecon.pdf>, 2003.
- [12] Stuart Cheshire and Marc Krochmal. Dns-based service discovery, 2004.
- [13] The gnutella protocol specification v0.4, document revision 1.2. http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf.
- [14] Bruce Schneier. http://www.schneier.com/blog/archives/2005/02/sha1_broken.html.
- [15] Ion Stoica et al., 2001.
- [16] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric, 2002.

- [17] Frank Dabek et al.
- [18] Martin Fiedler. Videoaufbereitung für peer-to-peer-videostreaming. Master's thesis, Technische Universität Chemnitz, 2005.
- [19] Postgresql: The world's most advanced open source database. <http://www.postgresql.org>.
- [20] Comprehensive perl archive network. <http://www.cpan.org>.
- [21] Python programming language. <http://www.python.org>.
- [22] Felix von Leitner. Scalable network programming - or: The quest for a good web server (that survives slashdot). <http://bulk.fefe.de/scalable-networking.pdf>, 2003.
- [23] Chemnitzer linux cluster. <http://www.tu-chemnitz.de/urz/clic/index.html>.
- [24] The r project for statistical computing. <http://www.r-project.org/>.

Abbildungsverzeichnis

3.1	Verschiedene Möglichkeiten des Loadbalancings	6
3.2	RTP-Header	8
3.3	Zeitliche Ordnung von RTP-Paketen	9
3.4	Zustandsübergänge RTSP	10
3.5	Verschiedene Möglichkeiten zur Verbesserung der Skalierbarkeit von RTSP	12
4.1	Organisationsformen in Peer-To-Peer-Netzen.	14
4.2	Gnutella: Ping	17
4.3	Gnutella: Query	17
4.4	Gnutella: Get	18
4.5	Gnutella: Push	18
4.6	Routing in Freenet	20
5.1	Datenfluss im konstruierten Peer-To-Peer-Netz	25
5.2	Datenstruktur zur Speicherung des Vorhandenseins der Access Units	32
5.3	Datenstruktur zur Speicherung der Verbindungen zwischen den Peers	33
5.4	Datenstruktur zur Speicherung der Verbindungen zu den Peers	34
6.1	Datenmodell für den Tracker	37
6.2	Zusammenspiel der Klassen im Proxy	38
6.3	Struktur von mplayer-gluе	39
7.1	Latenzzeit und übertragene Blöcke beim HTTP Streaming	42
7.2	Entwicklung der Latenzzeiten bei Streaming über RTSP	44
7.3	Cachedistanz	45
7.4	Anteil der Latenzzeiten für Access Units 0 und 1	50
7.5	Cachedistanz	53
7.6	Verschiedene Methoden zur Clusterung der Access Units	54
7.7	Durchschnittliche Latenzzeit der einzelnen Operationen für die C-Implementierung des Trackers	56
7.8	Anteil der Latenzzeiten bei FINDPROXY	56
7.9	Anteil der Latenzzeiten bei FOUNDPROXY	57
7.10	Anteil der Latenzzeiten bei AVAILABLE	57
C.1	Netzaufbau	67

Tabellenverzeichnis

3.1	Skalierungsverbesserungen für HTTP	7
5.1	Kosten der Operationen im idealen Tracker	35
7.1	Datenvolumen der einzelnen Trackeroperationen	46
7.2	Nutzdatenraten am Tracker	48
7.3	Anteil der Downloads, die weniger als 0.5 Sekunden zum Starten benötigten	50
7.4	Durchschnittliche Latenzzeiten für FINDPROXY bei Access Unit 0	51
7.5	Anteil der Cache-Treffer aufgeschlüsselt nach Peers	52
A.1	Anteil der Downloads, die weniger als 0.2 Sekunden zum Starten benötigten	60
A.2	Anteil der Downloads, die weniger als 0.3 Sekunden zum Starten benötigten	61
A.3	Anteil der Downloads, die weniger als 0.4 Sekunden zum Starten benötigten	61
A.4	Anteil der Downloads, die weniger als 0.5 Sekunden zum Starten benötigten	62
A.5	Anteil der Downloads, die weniger als 1.0 Sekunden zum Starten benötigten	62
A.6	Anteil der Downloads, die weniger als 2.0 Sekunden zum Starten benötigten	63

Selbständigkeitserklärung

Ich erkläre hiermit, daß ich die vorliegende Diplomarbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Einzigste Ausnahme ist der erste Abschnitt von Kapitel 2, welcher gemeinsam mit Martin Fiedler, der die begleitende Arbeit [18] verfaßt hat, geschrieben wurde.

Chemnitz, den 14. Juli 2005