Technische Universität Ilmenau

Fakultät für Informatik und Automatisierung
Institut für Praktische Informatik

Studienjahresarbeit

Video On Demand über ein Peer-To-Peer-Netzwerk unter besonderer Berücksichtigung der Kontroll-möglichkeiten des Videoanbieters

---

Bearbeiter: Alexander Bösecke

Betreuer: Thorsten Strufe

Inhaltsverzeichnis

1. Abstrakt

Als Grundlage dient ein klassisches Streaming-System, welches um P2P-Funktionalität erweitert wird um die Kosten für ein derartiges System zu minimieren. Parallel soll untersucht werden, in welchen Grenzen es dem Betreiber möglich ist die Kontrolle über das gesendete Material und das System an sich zu behalten.

2. Einführung

Die Rechenleistung heutiger Heimcomputer scheint geradezu unendlich und verleitet daher bei einzelnen Anwendungen zu sehr hohen Qualitätsansprüchen. Video-Anwendungen sind hier ein Parade-Beispiel, denn die DVD als Trägermedium setzt bereits einen relativ hohen Qualitätsstandard.
Der Erfolg von Filesharing-Systemen zeigt zudem, das Mensch bereit ist, sich die Videos aus dem Internet zu „beschaffen“.
Hier zeigen sich nun 2 wesentliche Probleme:

1. die Verzögerung durch den nötigen Download

2. der Rechteinhaber hat keinerlei Kontrollmöglichkeiten

Dies lässt sich prinzipiell durch ein Streaming-System lösen. Probleme entstehen dann aber beim Provider, denn dieser muss ein System bereitstellen, welches der Last (Bandbreite, Rechenleistung usw.) zu Spitzenzeiten gewachsen ist, die restliche Zeit aber zum großen Teil brach liegt. Das heißt die Kosten stehen in keinem Verhältnis zum zu erwartenden Gewinn. Die Kosten sind also zu minimieren um so ein System wirtschaftlich zu betreiben zu können.

Zur Lösung dieses Dilemmas wird in aktuellen Forschungsarbeiten häufig entweder ein System auf der Basis von Multicast oder ein Peer-to-Peer (P2P) System vorgeschlagen. Multicast ist zwar eine sehr elegante Variante, hat aber das Problem der fehlenden Unterstützung durch die Internetprovider. So funktioniert Multicast in der Regel lediglich innerhalb der providereigenen Netze. Deshalb wird in dieser Arbeit ein P2P System verwendet. Weiterhin werden die Möglichkeiten analysiert, mit denen der Anbieter das Systems und die Nutzung des gestreamten Materials kontrollieren kann.

Die Arbeit gliedert sich wie folgt aus:

In Kapitel 3 werden die Anforderungen an ein derartiges System beschrieben.
In Kapitel 4 folgt die Analyse der Problemstellung und darauf aufbauend das Design und der Entwurf des Modells.
Kapitel 5 befasst sich mit der konkreten Implementierung.
Kapitel 6 beschreibt alternative Einsatzmöglichkeiten, zeigt generelle Probleme und Lösungsansätze auf.

3. Anforderungen

Ein Nutzer soll ein Video anfordern und nach einer unvermeidlichen aber möglichst kurz zu haltenden Zeit dieses anschauen können. Da Videos in der Regel zu groß sind, um in dieser Zeit das Video komplett runterladen zu können, gilt das Prinzip „play-while-downloading“. [1]

Voraussetzung dafür ist, dass die reale Downloadrate beim Nutzer größer als die Bitrate des Videos ist. Falls beim Nutzer die nötige Downloadrate nicht erreicht werden kann, soll das Video „auf Vorrat“ heruntergeladen werden können, das heißt zum Beispiel der Nutzer entscheidet er will morgen ein bestimmtes Video sehen, also startet er heute den Download.

Um für den Anbieter die Kosten für ein derartiges System zu minimieren, sollen Nutzer motiviert werden dem System Ressourcen wie Speicherplatz und Netzwerkbandbreite zur Verfügung zu stellen. Das bedeutet also, der Nutzer soll sich am Downloadprozess aktiv beteiligen. Der Nutzer soll aber in jedem Fall die vollständige Kontrolle über sein System behalten.

Der Server beziehungsweise der Anbieter muss kontrollieren können, welche Videos im System angeboten werden.
Für einen kommerziellen Einsatz muss man die Forderung stellen, dass der Anbieter kontrollieren, was der Nutzer mit dem heruntergeladenen Video anstellt. Da die Erfüllung derselbigen den Rahmen dieser Arbeit weit übersteigen würde, wird sich der Autor hier auf eine Diskussion / Untersuchung der Möglichkeiten beschränken.

4. Architektur-Modell

Nach dem klassischen Client-Server-Prinzip stellt ein Server einen Dienst und die dazugehörigen Ressourcen bereit. Der Client nutzt den Dienst und damit die Ressourcen. Das Bereitstellen der Ressourcen kann für den Betreiber des Servers aber einen nicht abzuschätzenden Kostenfaktor bedeuten, insbesondere wenn die Anzahl der Clients stark schwankt. Wenn der Dienst hauptsächlich aus dem zur Verfügung stellen von Dateien besteht, bietet es sich an, die Ressourcen vom Server zu trennen. Die Ressourcen werden dann von so genannten Seeds bereit gestellt und der Server verwaltet die Ressourcen „nur“ noch. Wenn Client und Seed zusammenfallen, nennt man das Produkt in der Regel Peer. Dieser Peer nutzt also den Dienst und die Ressourcen, stellt aber gleichzeitig seine eigenen Ressourcen dem System zur Verfügung. Da auch bei diesem Modell der Server immer noch einen „single point of failure“ darstellt, kann man die Verwaltung der Ressourcen den Peers übertragen, der Dienst entsteht also durch den Zusammenschluss der Peers. Dabei ergeben sich aber zwei wesentliche Probleme – neue Peers müssen einen Zugangspunkt zum System finden und der nötige Traffic zum Verwalten der Ressourcen kann das System erheblich ausbremsen.

Hier wird ein Modell mit einem zentralen Server verwendet. Die Ressourcen werden von Seeds bereitgestellt, welche sowohl der Betreiber als auch die Nutzer des Dienstes betreiben.

Als erstes folgen einige Begriffsdefinitionen, danach die Beschreibung des Modells.

4.1. Definitionen

Der Server S kontrolliert die Ressourcen und ermöglicht den Dienst, das bedeutet unter anderem, dass er als zentrale Anlaufstelle für die Seeds und Clients dient. Weiterhin stellt er eine Liste aller aktuell verfügbaren Videos bereit und speichert zu jedem Video eine Reihe von Metadaten und zusätzlich von welchen Seeds Teile der Videos aktuell angeboten werden

Der Seed P[1] stellt Ressourcen (hier Bandbreite und Speicherplatz) zur Verfügung und kann die Nutzung dieser Ressourcen in einem gewissen Rahmen selbst kontrollieren, um zum Beispiel eine Überlastung zu vermeiden. Es wird zwischen Server-Seed und Client-Seed unterschieden; Server-Seeds dürfen dem Server beliebige Dateien anbieten, von Client-Seeds nimmt der Server nur Dateien an, für die bereits andere Seeds bereit stehen

Der Client C nutzt den Dienst und damit die angebotenen Ressourcen.

Clients und Seeds speichern zu jedem Video zusätzlich, zu den vom Server gelieferten Daten, die Liste der bereits erhaltenen Blöcke und das Video selbst.

Übersicht:

S: Server
L_V: Liste der Videos
L_B: Liste der Blöcke
L_P: Liste der Seeds
V: Video[2] (Medium, Datei)
R: Bitrate [kBit/s]

1 Die Begriffe Seed und Peer werden synonym verwendet, in den Quelltexten herrscht aus „historischen Gründen“ die Bezeichnung Peer vor.

2 Da es hier ausschließlich um Video-Streaming geht, werden die Begriffe Video und Datei synonym verwendet. Prinzipiell sind auch beliebige andere Medienformate (Audio, ...) denkbar.

G: Größe [MB]
A: Anzahl der Blöcke
B: Blockgröße
C: Client
D: mögliche Downloadrate [kBit/s]
Z: Pufferzeit [s]
P: Seed
U: mögliche Uploadrate [kBit/s]

4.2. Modell

Ziel ist es, die gegebene Downloadrate des Clienten optimal auszunutzen. Gleichzeitig sollen (müssen) die vorgegebenen Parameter des Videos eingehalten werden, das heißt im wesentlichen soll das Video mit der Bitrate ohne ruckeln / Verzögerung abgespielt wird.

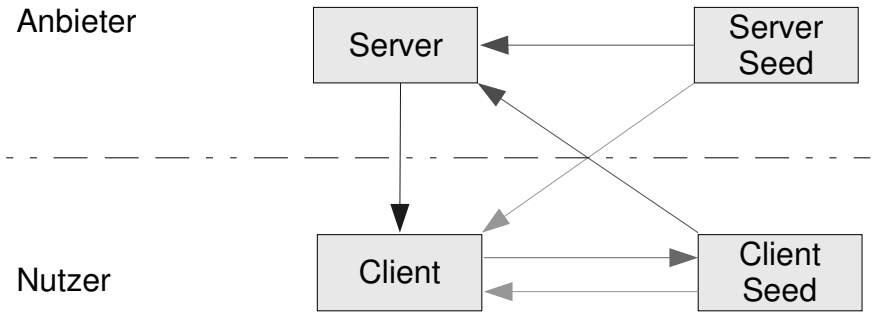
Zerlegung des Videos
Das Video wird in Blöcke gleicher Größe zerlegt. Seeds stellen immer komplette Blöcke zur Verfügung und Clients fordern immer komplette Blöcke beim Server an. Jeder Block wird wiederum in gleich große Segmente zerlegt. Der Client lädt die Blöcke segmentweise von den Seeds herunter.

Segmentgröße
Bei der Wahl der Segmentgröße ist ein Kompromiss zu finden:

- möglichst groß, um den Overhead durch die Kommunikation zwischen Client und Seeds zu minimieren
- nicht zu groß, damit zu langsame Seeds den Download nicht unnötig ausbremsen
- Beispiel: Pufferzeit Z = 30 s
  Wenn man festlegt, dass der Download eines Segmentes nicht länger als die halbe Pufferzeit dauern soll[1] und eine minimale Downloadrate von circa 1 kB/s erlaubt, ergibt sich eine Segmentgröße von 15 kB. Da die Blockgröße ein Vielfaches der Segmentgröße sein muss, sollte diese sinnvollerweise auf eine Zweierpotenz gerundet werden, also 16 kB/s.

Eine Orientierung der Segmentgröße an TCP-Paketgrößen erweist sich als nicht praktikabel, da der Datenanteil in TCP-Paketen nicht konstant groß ist und außerdem optimiert der TCP/IP-Stack des Betriebssystems die Pakete in der Regel selbstständig.

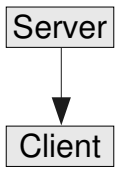
Systemstruktur
Folgendes Diagramm soll die Systemstruktur und den Informationsfluss zwischen den Komponenten verdeutlichen. Die Informationsflüße werden danach im Detail erklärt.



1 Um eine Reserve zu haben, falls der Download abgebrochen wird und das Segment ein zweites mal angefordert werden muss.

Informationsflüsse im Detail

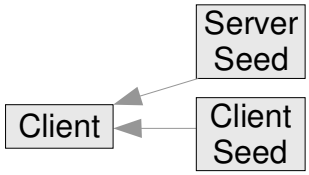
Der Client erfragt beim Server die folgenden Informationen:



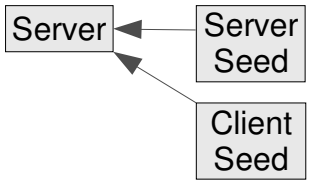
- Liste der angebotenen Videos
- Metadaten eines Videos
- bestimmte Anzahl von Blöcken und passend jeweils eine Liste von Seeds

---

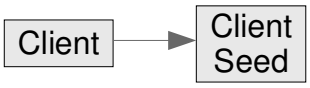
Der Client fordert bei mehreren Seeds gleichzeitig jeweils ein Segment an. Wenn der Download eines Segmentes nicht erfolgreich war, wird dieses Segment bei einem anderen Seed erneut angefordert.



Die Seeds teilen dem Server mit, welche Blöcke sie aktuell anbieten. Es ist nicht vorgesehen, dass der Server einen Einfluss darauf hat, was die Seeds speichern, insbesondere gibt es keinen Befehl zum Löschen / Speichern bestimmter Blöcke.



Der Client stellt die heruntergeladenen Blöcke dem Client-Seed zur Verfügung. Wenn Client und Client-Seed auf dem selben Rechner laufen, reicht dafür ein einfaches Event. Ansonsten ist ein aktives Übertragen der erhaltenen Daten erforderlich, dies ist aber ausdrücklich nicht Teil des Modells.



Downloadprozess

Der Download eines Videos erfolgt in 3 Phasen – Anfrage, Streaming, Seed.

Phase 1: Der Client fragt beim Provider nach einem Video, welches gestreamt werden soll und bekommt eine Reihe von Informationen wie Größe, Bitrate usw.

Phase2: Beim Start des Download fordert der Client beim Server eine bestimmte Anzahl von Blöcken einer Datei an und teilt dem Server dabei mit, welche Blöcke er bereits hat. Die Anzahl der Blöcke bestimmt sich durch die Pufferzeit.

Die folgenden Beispiele bauen aufeinander auf.

$$Pufferzeit: Z = 30\,s,\ Blockgröße: B = 1\,MB,\ Bitrate: R = 800\,kBit/s$$

$$Länge\,eines\,Blockes = \frac{B \times 1024 \times 8}{R} = 10{,}24\,s$$

$$Anzahl\,Blöcke = \left| \frac{Z}{Länge\,eines\,Blockes} \right| = 3$$

Am Anfang des Downloads fordert der Client demnach 3 Blöcke an. Danach fordert er immer so viele Blöcke an, wie er innerhalb der Pufferzeit runterladen kann. Dieser Wert wird nach jedem Download angepasst.

$$Downloadzeit_{alt} = 10\,s$$

$$Anzahl\ Blöcke_{neu} = \left| \frac{Z}{Downloadzeit_{alt}} * Anzahl\ Blöcke_{alt} \right| = 9$$

Der Server bestimmt, welche Blöcke der Client als nächstes herunterladen soll. Dadurch ist der Server in einem gewissen Rahmen in der Lage zu kontrollieren, wie sich die Blöcke im System verteilen. Dazu teilt der Client dem Server mit, welchen Block er gerade abspielt. Der Server ermittelt dann, welche Blöcke dem Client noch fehlen, damit er immer Material zum abspielen hat.

Annahme: Der Client spielt Block 10 ab und fordert neue Blöcke an. Im Idealfall spielt der Client Block 13 ab, wenn der Download dieser Blöcke abgeschlossen ist, die Blöcke 14-16 müssen demzufolge dann beim Client angekommen sein. Wenn nun der Client die Blöcke 11-14 bereits hat, muss der Server dem Client also die Blöcke 15 und 16 senden. Die restlichen 7 Blöcke verteilt der Server nach einem vorgegebenen Schema.[1]

Zu jedem Block wählt der Server eine Reihe von Seeds aus, welche für den Client am besten geeignet sind. Kriterien für diese Auswahl sind zum Beispiel:

- maximale Uploadrate des Seeds
- Schätzung der Anzahl der Clients, die gerade mit dem Seed verbunden sind
- geographische Nähe – IP-Address-Bereich (d. h. Clients im Telekomnetz erhalten bevorzugt Seeds im Telekomnetz)
- ...

Der Client fragt dann die Seeds nach einzelnen Segmenten, von denen er die höchsten Downloadraten erwarten kann. Die Wahl eines Seeds kann mit folgendem Algorithmus erfolgen:

- für jeden unbekannten Seed durchschnittliche Downloadrate = maximale Downloadrate
- wähle Seed mit höchster durchschnittlicher Downloadrate
- nach Download berechne neue durchschnittliche Downloadrate

Die neue durchschnittliche Downloadrate errechnet sich aus dem alten Durchschnitt ($Durchschnitt_{Alt}$), der Anzahl der Downloads ($ZahlDerDownloads$) und der Zeit, die der gerade abgeschlossene Download gedauert hat ($AktuelleZeit$). Die Berechnung erfolgt mit folgender Formel:

$$Durchschnitt_{Neu} = \frac{Durchschnitt_{Alt} * ZahlDerDownloads + AktuelleZeit}{ZahlDerDownloads + 1}$$

Damit Downloads in vergangener Zeit nicht zu starken Einfluss haben, wird die Anzahl der Downloads nur bis zu einem bestimmten Maximum gezählt:

$$wenn\ \ ZahlDerDownloads < Maximum: \quad ZahlDerDownloads + 1$$

Ein Problem dabei ist, dass der Client vom Server regelmäßig die gleichen Seeds zugewiesen bekommen muss, damit sich ein realer Durchschnittswert ergeben kann.

1 Es ist auch eine zufällige Verteilung dieser „Restblöcke“ möglich. Welche Verteilung sich als sinnvoll erweist, müssen ausführliche Belastungstest zeigen und hängt zudem von der konkreten Situation des Systems ab, siehe auch Kapitel 6.3

Die Gründe, warum sowohl Server als auch Client eine Auswahl an Seeds treffen[2], sind:

- für den Server ist es nahezu unmöglich die perfekt passenden Seeds zu finden
- für den Client ist es nicht sinnvoll aus mehreren 1000 Seeds auswählen zu müssen

Phase 3: Sobald der Client die angeforderten Blöcke erhalten hat, werden sie auf die Festplatte geschrieben. Damit stehen sie erstens zum anschauen und zweitens für den Client-Seed zur Verfügung.

2 Für weitere Anmerkungen zu diesem Thema siehe auch Kapitel 6.3 → Auswahl der Seeds

5. Implementierung

Das gewählte Netzwerkprotokoll ist TCP, da es den Vorteil einer integrierten Fehlerkontrolle bei der Übertragung bietet. UDP würde weniger Overhead erzeugen und die benötigte Rechenleistung ist geringer. Für das schauen von Videos ist es zudem häufig egal, ob einzelne Pakete fehlerhaft oder gar nicht übertragen werden. Aber da hier ein P2P System verwendet wird, müsste bei der Verwendung von UDP ein System zur Fehlerkontrolle (insbesondere die erneute Übertragung von fehlerhaften oder verloren gegangenen Paketen ist wichtig) aufgebaut werden. Sonst könnten sich die Fehler aufsummieren und bei Seeds der „x-ten“ Generation würde nichts brauchbares ankommen.

Als Programmiersprache wurde Python gewählt, da der ursprüngliche Ansatz zur Realisierung dieses Projektes die Verwendung von BitTorrent vorsah.[1] Dies wurde verworfen, weil eine Erweiterung von BitTorrent in diese Richtung nicht sonderlich praktikabel ist. Python hat aber mehrere Eigenschaften, welche für ein derartiges Projekt sehr günstig sind. So sind kurze Entwicklungszyklen möglich und man kann vollständig objektorientiert programmieren, ohne dazu gezwungen zu sein. Weiterhin laufen Python-Programme auf fast jedem Betriebssystem. Dieses Projekt wurde fast komplett objektorientiert aufgebaut, wesentliche Funktionen sind aus Gründen der Parallelisierbarkeit als Thread realisiert.

Client und Seed sind komplett getrennt implementiert, das heißt man kann einen Seed separat starten, dadurch kann ein Nutzer zum Beispiel auf mehreren Rechnern einen Seed betreiben. Der Client kann über seine Oberfläche einen Seed starten, ist dazu aber nicht gezwungen. Dies hat weiterhin den Vorteil, das für den Server-Seed die gleiche Implementierung wie für den Client-Seed verwendet wird.

Um ein blockieren von Threads zu vermeiden, nutzen alle Socket-Funktionen einen Timeout von 5 Sekunden.

Im folgenden werden die Module und die darin enthalten wichtigen Klassen und Funktionen kurz beschrieben.

`Allgmein.py`: Hier sind Konstanten, Funktionen und Klassen definiert, die im gesamten System benötigt werden.

Klasse `Socking`: Diese Klasse realisiert einen von Socket abgeleiteten Kommunikationsdienst. Sie bietet die Funktionen `sendData` und `recvData`, welche eine nachrichtenbasierte Übertragung implementieren. Das bedeutet eine Nachricht, welche mit `sendData` gesendet wird, kann garantiert von einem `recvData` empfangen werden. Die Nachrichten sind in Header und Daten unterteilt, diese werden mit einem '.' getrennt. Die Felder des Headers werden durch '-' getrennt, in der derzeitigen Implementierung gibt es 2 Felder – die Länge der Daten und der Typ der Daten. Es gibt folgende Datentypen:

- `MSG`: normale Nachricht
- `ERR`: Fehler
- `FIL`: Datei (Block, Segment)
- `ATH`: Authentifizierung
- `GET`: Anfragen
- `RPL`: Antworten auf `GET`

1 BitTorrent ist in Python geschrieben.

Alle Daten, welche nicht vom Typ MSG oder FIL sind, werden „gepickelt“. Dies ist eine Technik in Python, welche fast jedes Objekt in einen String verpacken kann.[1] Dieser String wird vom Empfänger in das ursprüngliche Objekt entpackt.
In dieser Arbeit sind Befehle in der Form `<Typ der Daten . Daten>` angegeben. Wenn `Daten` nicht in Hochkomma eingeschlossen ist, wird die entsprechende Datenstruktur aus dem Anhang verwendet, sonst wird der angegebene String gesendet.

Klasse `FileIndex`: Diese Klasse nutzen der Client und der Seed um die Metadaten der Videos zu speichern und auf diese zuzugreifen. Ein Seed, welcher über die Oberfläche eines Clienten gestartet wird, nutzt die selbe Instanz dieser Klasse wie der Client. Dadurch ist die nötige Kommunikation zwischen Client und Seed auf ein Minimum reduziert.
Die Metadaten selbst werden in dem Shelve `Index`[2] gespeichert. Zum eindeutigen Identifizieren einer Datei und damit als Schlüssel im Index wird ein Hashwert der Datei genutzt. Zum erzeugen des Hashwertes werden die md5 Hashwerte der einzelnen Blöcke der Datei aneinander gehängt.[3] Im folgenden beziehe ich mich mit hashString immer auf diesen Hashwert einer bestimmten Datei. Zu jeder Datei werden die folgenden Metadaten gespeichert:

- `BlockCount`: Anzahl der Blöcke für diese Datei
- `BlockList`: Liste der vorhandenen Blöcke
- `BitRate`: Bitrate des Videos[4]
- `FileSize`: Dateigröße
- `SavedSize`: Gesamtgröße der bereits heruntergeladenen Blöcke, dieser Wert ist lediglich geschätzt und dient ausschließlich der Anzeige in der Oberfläche des Clienten.
- `FileName`: Name der Datei, in welcher das Video gespeichert ist.
- `Complete`: Ist die Datei komplett?
- `Download`: Wird die Datei aktuell heruntergeladen?

Funktion `createIndex`: Diese Methode wird vom Konstruktor der Klasse aufgerufen und stellt sicher, dass die Daten im Index mit den real vorhandenen Videos übereinstimmen.

Funktion `getIndex`: Gibt den kompletten Index zurück. Nutzt der Seed, wenn er dem Server mitteilen will, dass er neue Blöcke speichert. Nutzt der Client, um seine Oberfläche zu aktualisieren.[5]

Funktion `getAttr`: Ermittelt ein bestimmtes Attribut einer Datei.

Funktion `remFile`: Entfernt eine Datei sowohl aus dem Index als auch auf der Festplatte.

Funktion `updateIndex`: Aktualisiert die Metadaten einer Datei beziehungsweise legt einen neuen Eintrag im Index für die Datei an. Setzt außerdem die Flags SavedSize und Complete und erzeugt die Events eventNewBlocks (für den Seed) und eventNewData (für den Client).

Funktion `hasBlock`: Ist eine Datei oder ein bestimmter Block einer Datei vorhanden?

1 Auch die Datenstrukturen welche im Anhang beschrieben sind, werden so verpackt und zwischen den Parteien ausgetauscht.
2 Ein Shelve ist eine Art Datenbankdatei, zur weiteren Definition siehe Kapitel 7.3.
3 Da die Hashwerte sehr lang werden können, ist eventuell ein anderes Verfahren zum Identifizieren der Dateien sinnvoll.
4 Derzeit ungenutzt, da die Bestimmung recht schwierig ist.
5 Hier ist noch einiges an Optimierungsbedarf – Problem ist, dass man in Python einem Event keine Daten mitgeben kann, ein Event ist ein einfaches Flag. Man könnte aber eine globale Liste zum Datenaustausch zwischen den verschiedenen Klassen nutzen.

`ClientData.py`: Dieses Modul enthält die „Arbeitsklassen“ des Clienten.

Klasse `ClientData`: Diese Klasse bietet die Funktionen für die Kommunikation mit dem Server an.

Funktion `connect`: Stellt eine Verbindung zum Server her und authentifiziert sich diesem gegenüber. Die Verbindung bleibt bestehen, bis sie explizit getrennt wird.

Funktion `disconnect`: Trennt die Verbindung zum Server.

Funktion `getFileIndex`: Holt vom Server die Liste aller angebotenen Dateien und übergibt an die Oberfläche des Clienten die Struktur ClientFileList.

Funktion `getFileInfo`: Holt vom Server die Metadaten zu einer bestimmten Datei und übergibt der Funktion updateIndex der Klasse FileIndex die Struktur FileInfo.

Funktion `getFile`: Startet einen Thread, um die angegebene Datei herunterzuladen. Dieser Thread ist in der Klasse getFileThread implementiert.

Klasse `GetFileThread`: In Python gibt es im wesentlichen 2 Methoden einen Thread zu erzeugen: Erstens mit der Funktion „start_new_thread“, welcher die auszuführende Funktion als Parameter zu übergeben ist. Und zweitens mit einer Klasse welche von „threading.Thread“ abzuleiten ist. Die Funktion start dieser Klasse startet den Thread, welcher in der Funktion run implementiert wird.

Funktion `run`: Die Downloadprozesse werden mit der globalen Liste getFileList gesteuert. Wenn der Download einer Datei startet, wird der Dateiname an diese Liste angehängt, der Download stoppt, sobald der Dateiname nicht mehr in dieser Liste steht.[1]

Der Download läuft bis für die Datei das Flag Complete gesetzt ist.[2] Zuerst wird die aktuelle BlockListe dem Server gesendet. Der Server antwortet mit einer Liste von Blöcken mit dazu zugehörigen Seeds, welche als nächstes geholt werden sollen. Der Download dieser Blöcke erfolgt in 3 Phasen – Initialisierungsphase, Download-Phase und Post-Download-Phase.

In der Initialisierungsphase werden folgende Daten erzeugt:

- `allSeg`: Gesamtzahl der Segmente
- `haveSeg`: Zahl der bereits heruntergeladenen Segmente, beim Start 0
- `BlockData`: Dictionary, welches als Schlüssel die Blöcke hat. Die Werte sind jeweils eine Liste mit 2 Elementen: erstes Element ist die Liste der Seeds, zweites Element ein Dictionary, welchem die Segmentnummern als Schlüssel zugewiesen werden. Diesem Dictionary werden die empfangenen Segmente als Wert übergeben.
- `SegList`: Liste von Tupeln aller Segmente in der Form (Block, Segment)

Download-Phase: Die Anzahl der gleichzeitigen Downloads wird durch eine globale Semaphore gesteuert. Wenn der Zähler der Semaphore null ist, wird gewartet. Das heißt die Zahl der Seeds, von denen gleichzeitig Segmente heruntergeladen werden können, ist auch dann konstant, wenn mehrere Videos parallel heruntergeladen werden. Den eigentlichen Downloadvorgang soll folgender Pseudocode verdeutlichen:

1 Wenn in Oberfläche des Clienten „Download stoppen“ ausgewählt wird, wird einfach der Dateiname aus der Liste entfernt, wenn alle Downloads stoppen sollen, wird der Liste eine leere Liste zugewiesen.

2 Zur Erinnerung: dieses Flag wird von der Funktion updateIndex der Klasse fileIndex automatisch gesetzt.

```
Solang allSeg != haveSeg und Dateiname in getFileList:
   wenn SegList nicht leer:
      fordere Semaphore an
      nimm erstes Element von SegList
      wenn für den gewählten Block die Liste der Seeds nicht leer ist:
         ermittle einen Seed mit getPeerAddr
         starte getBlockPart mittels start_new_thread
      sonst:
         gib Semaphore frei
         hänge das Segment an das Ende von SegList an
   sonst:
      warte bis allSeg == haveSeg oder SegList nicht leer ist
```

Post-Download-Phase: Wenn der Download nicht abgebrochen wurde, der Dateiname also noch in der Liste getFileList steht, werden die heruntergeladenen Blöcke an die BlockListe gehängt und die Funktion writeFile wird aufgerufen.

Funktion `getPeerAddr`: Wählt für einen gegebenen Block aus der vom Server für diesen Block gelieferten Liste einen Seed aus. Der gewählte Seed wird aus der Liste entfernt, damit ein Seed nicht gleichzeitig zweimal gefragt wird.

Funktion `getBlockPart`: Holt ein Segment eines Blockes von dem gegebenen Seed. Wenn die Übertragung erfolgreich war, werden die empfangenen Daten des Segmentes in das Dictionary BlockData geschrieben. Falls die Übertragung fehlgeschlagen sein sollte, wird das Segment wieder an SegList angehängt.
Der Seed wird an die passende Liste in BlockData gehängt und das Semaphore wird freigegeben.

Funktion `writeFile`: Ermittelt für jedes Segment in BlockData die Position in der Datei und schreibt die Daten des Segmentes an diese Stelle.

`ClientGui.pyw`: Der Client wird über eine graphische Oberfläche bedient, welche mittels wxPython[1] programmiert ist. Die Oberfläche ist komplett von der eigentlichen Programmlogik getrennt, das heißt es sollte ohne Probleme möglich sein, eine alternative Oberfläche zu entwickeln. In diesem Modul ist diese graphische Oberfläche des Clienten implementiert. Da in diesem Projekt aber nicht die Bedienung im Vordergrund steht, soll darauf nicht weiter eingegangen werden.

`Peer.py`: Hier ist der Seed implementiert.

Klasse `StreamSeed`: Wenn der Seed separat gestartet wird, erfüllt diese Klasse eine ähnliche Aufgabe wie beim Server die Klasse StreamServer. Das heißt der Thread für den eigentlichen Seed wird gestartet; danach wird auf die Eingabe 'exit' am Prompt gewartet und daraufhin das Event eventStopPeer gesetzt.

Klasse `StreamingPeer`: Diese Klasse ist von threading.Thread abgeleitet und bildet den eigentlichen Kern des Seeds. Der Konstruktor erzeugt einen Socket für die Anfragen der Clienten. Außerdem wird bei Bedarf eine Instanz der Klasse FileIndex erzeugt.[2]

Funktion `run`: Wenn das Event eventNewBlocks gesetzt ist, wird die Funktion sendFileIndex aufgerufen. Weiterhin wird auf Requests von Clients gewartet und die Funktion sendFile als extra Thread gestartet.

1 Python-Wrapper für wxWindows, eine plattformübergreifende Bibliothek für graphische Oberflächen
2 Bedarf besteht wenn der Seed nicht über die Oberfläche eines Clienten gestartet wird. Sonst wird die entsprechende Instanz des Clienten genutzt.

Funktion `sendFileIndex`: Meldet sich entweder beim Server an und sendet die SeedFileList oder meldet sich beim Server ab. Die Verbindung wird nach dem senden der SeedFileList wieder beendet, um die Last für den Server zu minimieren.

Funktion `sendFile`: Es wird anhand der Blocknummer und der Segmentnummer die Position des gewünschten Segmentes im Video bestimmt, das Segment wird gelesen und an den Client übertragen. Der Client erhält eine Fehlermeldung, falls er ein ungültiges Segment angefordert hat.

`Server.py`: Hier ist der Server implementiert.

Klasse `StreamServer`: Mit dieser Klasse wird der Server gestartet. Der Konstruktor startet als separaten Thread den Dispatcher und wenn gewünscht einen Seed. Danach geht er in eine Schleife und wartet das am Prompt „`exit`“ eingegeben wird, worauf die Events `eventStopPeer` und `eventExitApp` gesetzt werden, was den Dispatcher und gegeben falls den Seed veranlasst sich zu beenden.

Klasse `Dispatcher`: Die einzige Aufgabe dieser Klasse ist es, Verbindungsanfragen von Clients und Seeds entgegen zunehmen und jeweils einen Thread HandleClient zu starten.

Klasse `HandleClient`: Die Clients und Seeds authentifizieren sich mit `<ATH . 'Seed'>` beziehungsweise `<ATH . 'Client'>`[1] und die entsprechende Funktion client oder seed wird gestartet.

Funktion `client`: Der Server wartet, bis der Client eine konkrete Anfrage mittels `<GET . 'Anfrage'>` stellt. Wenn eine Anfrage kommt, wird jeweils die entsprechende Funktion der Klasse sharingFiles aufgerufen. Diese Funktion liefert eine Datenstruktur zurück, welche dem Client mit `<RPL . Datenstruktur>` gesendet wird. 3 Anfragen mit jeweils 2 Parametern sind möglich:

Anfrage	Funktion	Parameter	
`FileIndex`	`getFileIndex → clientFileIndex`	`dummy`	`dummy`
`FileInfo`	`getFileInfo → FileInfo`	`hashString`	`dummy`
`File`	`getPeer → BlockData`	`hashString`	`Blockliste des Clienten`

hashString bezieht sich auf die gewünschte Datei. Blockliste ist Liste der Blöcke, welche der Client bereits hat. Die „dummy“-Parameter können beliebige Werte annehmen, der Grund für ihre Verwendung ist eine Vereinfachung des Programmcode.
Die Verbindung wird getrennt, sobald der Client <MSG 'exit'> sendet.

Funktion `seed`: Der Server erfragt beim Seed mit `<GET . 'SharedFiles'>` welche Dateien dieser anbietet. Der Seed antwortet mit `<RPL . SeedFileList>`. Wenn `SeedFileList` leer ist, wird die Funktion `remPeer` der Klasse `SharingFiles` aufgerufen, ansonsten `addPeer`.

Klasse `SharingFiles`: Diese Klasse implementiert im wesentlichen die Struktur `SharedFiles` und bietet die Funktionen, mit denen der Server darauf zugreift.

Funktion `addPeer`: Wenn ein Seed dem Server die Liste der angebotenen Blöcke schickt, wird diese Funktion aufgerufen. Wenn die entsprechenden Dateien bereits in dem Dictionary `SharedFiles` stehen, wird die `SeedAdresse` an die Liste der Seeds bei den jeweiligen Blöcken gehängt. Wenn die Dateien noch nicht exis-

1 Hier dient die Authentifizierung nur dazu, um Clients und Seeds unterscheiden zu können.

tieren und die Funktion mit dem Parameter `Server=1` aufgerufen wurde, werden die Dateien mit den Metadaten dem Dictionary hinzugefügt.

Funktion `remPeer`: Wenn ein Seed sich beim Server abmeldet, werden die Listen der Seeds nach der `SeedAdresse` durchsucht und diese gegebenenfalls entfernt.

Funktion `getFileIndex`: Diese Funktion liefert das Dictionary `ClientFileList` zurück.

Funktion `getFileInfo`: Diese Funktion liefert zu einem gegebenen `hashString` das Dictionary `FileInfo`.

Funktion `getPeer`: Diese Funktion wählt eine Anzahl von Blöcken und dazu jeweils eine Anzahl von Seeds aus. Da derzeit noch unklar ist, nach welchen Kriterien die Auswahl der Seeds erfolgen soll, werden jeweils alle passenden Seeds gewählt. Die Auswahl der Blöcke erfolgt nach dem in Kapitel 4.2 beschriebenen Verfahren.

`ViewMovie.py`: Dieses Modul ist eine rudimentäre Implementierung eines Videoplayers. Dieser setzt auf pyGame auf, deshalb ist er auf das MPEG-Format beschränkt, aber da in dieser Arbeit das verwendete Videoformat nur eine untergeordnete Rolle spielt, soll auch darauf nicht weiter eingegangen werden.

6. Analyse

6.1. Kontrollmöglichkeiten

Bei der Kontrolle des Anbieters über das System im allgemeinen und den Nutzer im speziellen sind verschiedene Aspekte zu beachten, die im folgenden aufgezeigt werden sollen.

1. Videoangebot: Durch die Unterscheidung der Seeds in Server-Seeds und Client-Seeds, ist es dem Betreiber möglich zu kontrollieren, welche Videos in dem System verbreitet werden. Das heißt es ist dem Nutzer nicht möglich vom Anbieter nicht genehmigte Videos anzubieten. Wenn die Videos mit einer digitalen Signatur (Wasserzeichen oder ähnliche Verfahren) versehen werden, kann sichergestellt werden, dass der Nutzer nur Videos anbieten kann, die über das System bezogen wurden.
2. Videonutzung
   Damit der Anbieter die Nutzung der Videos kontrollieren kann, müssen die Videos derart abgespeichert werden, dass sie außerhalb der Software für den Client / Client-Seed nicht nutzbar ist. Das bedeutet im wesentlichen eine verschlüsselte Speicherung der Videos. Durch die Verschlüsselung sind zudem unterschiedliche Nutzungsmodelle realisierbar.[1]
   Das Problem ist, dass die Videos für die Nutzung zumindest teilweise entschlüsselt werden müssen. Was mit dem entschlüsselten Teil des Videos geschieht, kann der Anbieter kaum noch kontrollieren, da der Nutzer auf seinem Rechner tun und lassen kann was er will. Es gibt zwar aktuell einige Entwicklungen, die dem Anbieter diese Kontrolle ermöglichen würden, aber diese schränken den Nutzer in seinen Rechten erheblich ein.
3. Sabotage
   An dieser Stelle geht es darum, dass ein Seed vorgibt bestimmte Blöcke anzubieten, in Wirklichkeit aber Segmente versendet, welche nicht Teil der Blöcke sind. Verhindern lässt sich dies nicht, aber da dem Client die Hash-Werte der Blöcke zur Verfügung stehen, kann er erkennen, dass er ein falsches Segment erhalten hat. Da auf diese Weise nicht erkannt werden kann, welches Segment falsch ist, bleibt dem Client nichts anderes übrig als beim Server für den betroffenen Block andere Seeds anzufordern.
   Es ist möglich, das der Server zusätzlich die Hash-Werte der Segmente speichert und diese dem Clienten bei Bedarf zusendet. Dies würde aber einen nicht unerheblichen Aufwand bedeuten, der sich nur selten rentiert.

6.2. Alternative Einsatzmöglichkeiten

Das hier vorgestellte System ist so flexibel gestaltet, dass sich durch geringe Änderungen der Struktur weitere Einsatzfelder eröffnen.

1. Lastverteilung ohne Einbeziehung des Nutzers
   Wenn das Ziel eine dynamische Lastverteilung ist, aber der Nutzer nicht in den Download-Prozess einbezogen werden soll, kann auf die Client-Seeds verzichtet werden. Der Anbieter ist damit in der Lage auf Lastveränderungen sehr flexibel zu reagieren. Einmal kann das Angebot auf den Seeds der Situation anpassen werden und zum anderen sind zusätzliche Seeds relativ einfach in das System zu integrieren.

1 zum Bsp.: das Video darf nur x-mal oder nur in einem bestimmten Zeitraum angeschaut werden

2. dezentrale Struktur
   Wenn das Ziel eine Erhöhung der Ausfallsicherheit des Systems ist, besteht die Möglichkeit mehrere Server parallel zu betreiben. Die Server tauschen ihre Informationen[1] regelmäßig aus, so dass jeder Server ein Bild über das komplette System hat. Dieses Bild ist zwar vollständig aber nicht immer aktuell.
   Alternativ können auch die Anfragen der Clients an die anderen Server weitergeleitet werden. Die Server antworten dann dem Clienten entweder direkt oder der Ursprungsserver sammelt die Antworten und sendet diese dem Clienten. Im ersten Fall können die Server aber keine Vorauswahl an Seeds treffen, im zweiten dauert es wesentlich länger bis der Client eine Antwort erhält.
   Welche Variante besser geeignet ist, müssten umfangreiche Tests zeigen.
3. System ohne Kontrolle
   Um ein System ohne Kontrolle eines Anbieters zu realisieren, kann die Unterscheidung der Seeds weggelassen werden, dass heißt jeder Seed kann beliebige Videos anbieten.
   Zusätzlich kann die Funktionalität des Server in den Seed integriert werden. Die Clienten melden sich dann bei einem lokalen Seed an.[2] Die Anfragen des Clienten werden an andere Seeds weitergeleitet, welche die Anfragen wiederum an ihnen bekannte Seeds weiterleiten. Der Ursprungs-Seed sammelt die Antworten und leitet sie an den Clienten weiter. Um die Antwortzeiten zu verkürzen, können die Seeds einer begrenzten Zahl von anderen Seeds mitteilen, welche Video-Blöcke sie anbieten.
4. Live-Streaming
   Die Videos bekommen ein zusätzliches Attribut, das aussagt, ob es sich um ein Live-Video handelt. Live-Video bedeutet, dass nicht alle Blöcke des Videos von Anfang an zur Verfügung stehen und unter Umständen sogar, dass die Anzahl der Blöcke nicht bekannt ist.
   Für den Client bedeutet dies, dass er während des Downloadprozesses regelmäßig über die Anzahl der Blöcke informiert werden muss, damit er weiß wann der Download abgeschlossen ist. Als Problem könnte es sich erweisen, wenn der Client meint den Download abgeschlossen zu haben, weil er alle Blöcke hat, die dem Server aktuell bekannt sind, aber zu einem späteren Zeitpunkt noch weitere Blöcke hinzukommen.
   Der Seed wird mit dem Event `eventNewBlocks` über neue Blöcke informiert. Es muss also eine zusätzliche Komponente implementiert werden, welche den Index in der Klasse `FileIndex` aktualisiert.

6.3. Was noch zu tun ist

Dem in Kapitel 4 beschriebenen Modell fehlt es noch an der konkreten Bestimmung einiger Parameter, diese können zum großen Teil jedoch erst durch ausführliche Performanztests ermittelt werden.
Die letzten beiden Punkte sind nicht bearbeitet worden, da dies erstens den Umfang dieser Arbeit deutlich sprengen würde und zweitens dem Autor die Kenntnisse dazu fehlen.

1. Pufferzeit
   Eventuell kann es sich als sinnvoll erweisen, die Pufferzeit von der Länge des Videos anhängig zu machen, denn der Nutzer ist sicher bereit auf einen über 2 Stunden langen

1 Informationen darüber, von welchen Seeds die Blöcke aktuell angeboten werden
2 Es muss nicht unbedingt ein lokaler Seed sein, dies entbindet aber den Clienten davon einen verfügbaren Seed zu suchen. Ein Seed würde dann auch nur Anfragen von einem lokalen Clienten annehmen.

Film länger zu warten als auf einen maximal 10 minütigen Videoclip.
Die Pufferzeit darf aber nicht zu kurz werden, da dies den Kommunikationsoverhead erhöht und damit die zu erwartende Downloadrate für den Client sinkt. Auf der anderen Seite darf die Pufferzeit nicht zu lang sein, damit der Client durch die Anforderung einer unterschiedlichen Anzahl von Blöcken flexibel auf Schwankungen in der Downloadrate reagieren kann.

2. Bestimmung der Bitrate
   Um eine möglichst geringe Videogröße und gleichzeitig eine hohe Qualität zu erreichen, verwenden viele aktuelle Videocodecs eine variable Bitrate. In diesem Fall ist entweder die Blocklänge oder die Blockgröße variabel. Eine feste Blocklänge hat den Vorteil, dass die Anzahl der Blöcke innerhalb der Pufferzeit konstant ist. Eine variable Blockgröße hat aber den entscheidenden Nachteil, dass die Downloadzeit für einzelne Blöcke nicht mehr vorhersehbar ist – eine feste Blockgröße ist also zwingend erforderlich. Das Problem reduziert sich demnach auf die Bestimmung der Blocklänge, um die Anzahl der Blöcke innerhalb der Pufferzeit bestimmen zu können. Die Blocklänge lässt sich aber nur in Abhängigkeit vom Videocodec bestimmen und dazu ist keine Entscheidung getroffen worden.[1]

3. Auswahl der Seeds
   Dieser Punkt ist im Modell nur sehr oberflächlich behandelt und implementiert ist in diesem Zusammenhang noch gar nichts.[2]
   Der Client sollte für den Download die Seeds wählen, von denen er die höchsten Downloadraten erwarten kann. Das wesentliche Problem dabei ist – woher soll der Client das Wissen nehmen, welche Seeds für ihn günstig sind? Das Mitteln der Downloadraten über einen längeren Zeitraum kann funktionieren, kann bei einer großen Zahl von verfügbaren Seeds aber auch sehr ineffizient sein.

   Daher soll an dieser Stelle die folgende Idee präsentiert werden.
   Damit nicht jeder Client die Downloadraten zu den Seeds selbst ermitteln muss, tauschen die Clients diese Daten in einem eigenem P2P-Netz aus. Dazu muss jeder Client den Informationen der anderen Clients einen Wert zuordnen, der die Relevanz für die eigene Downloadrate angibt.
   Der Vorteil ist, dass für der Server bei der Vorauswahl an Seeds deutlich weniger Aufwand treiben muss. Der Client muss die zu erwartenden Downloadraten nicht mehr selbst ermitteln und kann daher aus einer größeren Anzahl von Seeds sinnvoll wählen.
   Als Nachteil kann sich herausstellen, dass man unter Umständen das Problem nur verlagert und (unnötigen) Overhead produziert.

4. Auswahl der Blöcke
   Zu diesem Punkt ist bereits einiges gesagt worden, aber um dieses Problem zu lösen, muss erst die Blocklänge / Blockgröße zuverlässig bestimmt werden können.

5. Identifikator
   Die derzeit verwendete Methode zum eindeutigen Identifizieren der Videos kann sich bei großen Videos als sehr ungünstig erweisen, da der Schlüssel direkt von der Länge des Videos abhängt.
   Eine Alternative ist, dass der Anbieter einfach eindeutige Schlüssel für die Videos vergibt. Das ist aber nur dann sinnvoll möglich, der Anbieter die Server-Seeds von einer zentralen Stelle aus konfigurieren und überwachen kann.

1 Die Intention des Autors ist es zudem, dass das System unabhängig vom verwendeten Videocodec funktioniert.
2 Aktuell bekommt der Client alle verfügbaren Seeds mitgeteilt und der Client wählt für den Download einfach den ersten aus der Liste aus.

6. Zahl der parallelen Downloads / Uploads
   Wenn man von unsymmetrischen Netzwerkverbindungen[1] ausgeht, kann ein einzelner Seed die Downloadrate eines Clienten nicht ausnutzen. Ein Client muss demnach von mehreren Seeds gleichzeitig Daten herunterladen. Gleichzeitig macht es aber keinen Sinn einen schnellen Seed auszubremsen nur um eine bestimmte Zahl von Downloads parallel durchführen zu können. Die Zahl der parallelen Downloads muss sich also an der aktuellen Auslastung der Netzwerkverbindung orientieren. Auf der Seite der Seeds stellt sich das Problem in ähnlicher Weise – der Seed sollte Anfragen von Clients zurückweisen, wenn er ausgelastet ist.

7. Creditsystem
   Ein Creditsystem ist nötig, weil es für die Nutzer einen Anreiz geben muss, als Seed Ressourcen bereitzustellen.
   Credits sind eine Art virtuelle Währung und können einen Gegenwert in realer Währung haben.[2] Wenn ein Client ein Video runterladen will, so bezahlt er dieses mit Credits. Im Gegenzug bekommt der Seed für jedes erfolgreich übertragene Segment Credits gutgeschrieben. Um einen Missbrauch zu vermeiden, sollten sowohl die Clients als auch die Seeds dem Server[3] mitteilen, wer wieviele (bzw. welche) Segmente übertragen hat.
   Als Abrechnungssystem bietet sich eine Micropayment-Lösung an. [2]

8. Authentifizierung
   Dieser Punkt steht in direktem Zusammenhang mit dem Creditsystem. Eine zuverlässige Authentifizierung ist notwendig, damit Clients und Seeds eindeutig einem Nutzer zugeordnet werden können. Weiterhin muss der Server die Server-Seeds und Client-Seeds unterscheiden können. Für das Creditsystem und eventuell auch für die Seedauswahl muss der Client die Seeds erkennen können.
   Dafür kann zum Beispiel ein Public-Key Verfahren genutzt werden.

1 Die Downloadrate also wesentlich höher als die Uploadrate ist. Das beschriebene Problem existiert durch verschiedene Faktoren aber auch bei symmetrischen Netzwerkverbindungen.
2 Für einen nichtkommerziellen Einsatz ist zum Beispiel eine Upload-Download-Ratio für den Nutzer denkbar.
3 bzw. der Instanz, die das Creditsystem implementiert

7. Anhang

7.1. Literatur / Links

[1] D. Xu, M. Hefeeda, S. Hambrusch and B. Bhargava, „On Peer-to-Peer Media Streaming“, International Conference on Distributed Computing Systems 2002

[2] S. Micali, R. Rivest, „Micropayments Revisited“, Proceedings of the Cryptographer's Track at the RSA Conference 2002

7.2. Links

BitTorrent:
The Official BitTorrent Client - http://bitconjurer.org/BitTorrent/
BitTorrent protocol specification - http://bitconjurer.org/BitTorrent/protocol.html
Azureus – Java BitTorrent Client - http://azureus.sourceforge.net/

Trusted Computing:
Trusted Computing Group (TCG) - https://www.trustedcomputinggroup.org/home
Trusted Computing FAQ (deutsch) – http://moon.hipjoint.de/tcpa-palladium-faq- de.html
Microsoft's Next-Generation Secure Computing Base (NGSCB) -
http://www.microsoft.com/resources/ngscb/default.mspx
Informationen zur LaGrande-Technologie von Intel -
http://www.heise.de/newsticker/result.xhtml?url=/newsticker/meldung/40352

Python:
Python – www.python.org
wxPython – www.wxpython.org
wxWidgets – www.wxwidgets.org
pyGame – www.pygame.org

7.3. Datenstrukturen

einfache Datentypen: Integer, Long Integer, String, Boolean

Auf Rechnern mit 32 Bit Arithmetik ist Integer 32 Bit groß, Long Integer ist lediglich durch den Speicher in der Größe eingeschränkt. String und Boolean zeigen das allgemein erwartete Verhalten.

komplexe Datentypen: Liste, Tupel, Dictionary

Listen und Tupel sind Sequenzen von beliebigen Objekten, auf die Unterschiede soll hier nicht weiter eingegangen werden. Tupel werden in runde Klammern eingeschlossen, Listen dagegen in eckige.
Dictionaries sind ungeordnete Sequenzen von Schlüsseln und dazugehörigen Werten. Werte können ein beliebiges Objekt, Schlüssel können fast jeden Typs sein.
Die Notation von Dictionaries: {Schlüssel: Wert, }
Eine Ausnahme stellt die Struktur Index dar, diese ist ein Shelve, eine Art Datenbankdatei. Die Daten in einem Shelve sind ähnlich einem Dictionary organisiert, es gibt also Schlüssel-Werte-Paare. Die Schlüssel müssen Strings sein, die Werte können jedes Object sein, welches gepickelt werden kann.

Folgende Datenstrukturen werden in den jeweiligen Modulen verwendet:

allgemein:

```
hashString     = string
SeedAdresse    = (Host, Port)
```

Server:

```
SharedFiles    = {hashString: {BlockCount: integer,
                               BlockList:  {Block:[SeedAdresse]},
                               FileName:   string,
                               FileSize:   integer,
                              }
                  }
```

Seed / Client:

```
Index          = {hashString: {BlockCount: integer,
                               BlockList:  [integer, ],
                               FileName:   string,
                               FileSize:   long integer,
                               SavedSize:  integer,
                               Complete:   boolean,
                               Download:   boolean,
                              }
                  }
```

Seed:

```
SeedFileList   = {hashString: {BlockCount: integer,
                               BlockList:  [integer, ],
                               FileName:   string,
                               FileSize:   long integer,
                              }
                  }
```

Client:

```
ClientFileList = {hashString: FileName}
FileInfo       = {FileName:   string,
                  BlockCount: integer,
                  FileSize:   long integer
                 }
BlockData      = {Block: [SeedAddr, ]}
```

7.4. Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung erlaubter Hilfsmittel und der angegebenen Literatur angefertigt habe.

Ilmenau, .. Juli 2004, Alexander Bösecke