Systeme für Hochleistungsrechnen
Seminar SS 2003

Mike Sibler, Tobias Kluge,
Lars Bauer, Philipp Mergenthaler

Herausgeber:
Florin Isaila, Guido Malpohl
Vlad Olaru, Jürgen Reuter[a]

[a]Universität Karlsruhe
Institut für Programmstrukturen und Datenorganisation (IPD)
Lehrstuhl Tichy
Am Fasanengarten 5, 76128 Karlsruhe
e-Mail: {florin, malpohl, olaru, reuter}@ipd.uka.de

Zusammenfassung. Systeme für Hochleistungsrechnen sind Parallelrechner, die eingesetzt werden, wenn die Rechenleistung herkömmlicher Einzelprozessorsysteme nicht ausreicht. Die früher verwendeten, eng gekoppelten Multiprozessorsysteme werden, dem Trend zur globalen Vernetzung folgend, zunehmend durch preiswertere, lose gekoppelte Rechnerverbünde aus Standardrechnerknoten und Massenspeichern ersetzt. Die lose Kopplung ergibt vielfältige neue Herausforderungen in der Koordinierung zwischen den Rechnerknoten wie auch innerhalb jedes Knotens, um die Ressourcen im Verbund effizient nutzen zu können. Dies betrifft die koordinierte Zuteilung von Prozessoren und Speicher auf Prozesse ebenso wie die selbstorganisierende Abstimmung der Kommunikation zwischen den Knoten unter Berücksichtigung der Verbundtopologie. Vielfältige aktuell diskutierte Lösungsansätze von der Hardwareschicht über das Betriebssystem bis zur Anwendungsschicht werden in einer Reihe von Beiträgen, die im Rahmen des Seminars „Systeme für Hochleistungsrechnen“ im Sommersemester 2003 erarbeitet wurden, aufgezeigt und erörtert.

Vorwort

Im Sommersemester 2003 wurde im Seminar „Systeme für Hochleistungsrechnen“ aktuellen Trends aus Forschung und Entwicklung im Bereich der Hochleistungsrechner nachgespürt. Dazu wurde eine Reihe aktueller Themen aus den folgenden Gebieten angeboten:

- Cluster Computing
- Hardware-Architekturen
- Ad-hoc-Netzwerke
- Selbstorganisierende Netzwerke
- Peer-To-Peer-Netzwerke
- Speicherverwaltung in Betriebssystemen
- Parallele und verteilte Dateisysteme

Jeder Teilnehmer wählte hieraus ein Thema, um darüber in der Form eines medial gestützten Vortrages zu referieren. Um allen Teilnehmern die Gelegenheit zu geben, aus diesem Seminar nachhaltig etwas mitzunehmen, fertigte jeder Vortragende eine allen zugängliche schriftliche Ausarbeitung an. Die Ausarbeitungen finden sich in leicht redigierter Fassung durch die Editoren im vorliegenden technischen Bericht wieder. Es sind dies im Einzelnen:

- **Disk Scheduling**
  von Lars Bauer
  (Betreuer: Jürgen Reuter)
- **Gang Scheduling**
  von Philipp Mergenthaler
  (Betreuer: Jürgen Reuter)
- **Novel Network Systems**
  von Mike Sibler
  (Betreuer: Vlad Olaru)
- **Internet-Wide Storage**
  von Tobias Kluge
  (Betreuer: Florin Isaila)

Dieser Seminarband ist auch in elektronischer Form unter *http://www.ipd.uka.de/Tichy/* verfügbar.

Florin Isaila
Guido Malpohl
Vlad Olaru
Jürgen Reuter

Inhaltsverzeichnis

Vorwort iii

Einleitung 1

Kapitel 1. Disk Scheduling 3
1. Einleitung 3
2. Grundlagen 3
3. Aktuelle Disk-Scheduling-Verfahren 8
4. Ergebnisse 16

Literaturverzeichnis 19

Kapitel 2. Gang Scheduling 21
1. Einleitung 21
2. Gang Scheduling 22
3. Evaluierung von Gang-Scheduling bei feinkörniger Kommunikation 25
4. Ein verbesserter Algorithmus 29
5. Einsatz von Gang-Scheduling in der Praxis 32
6. Zusammenfassung 34

Literaturverzeichnis 35

Kapitel 3. Novel Network Systems
Overlay Netzwerke & Peer-to-Peer Systeme 37
1. Einleitung 37
2. Overlay Netzwerke 38
3. Peer to Peer (P2P) 44
4. Fallbeispiel für Collaboration: Pastiche 61

Literaturverzeichnis 69

Kapitel 4. Internet Wide Storage – Ein Einblick in Moderne Verteilte Dateisysteme 71
1. Motivation 71
2. Überblick 72
3. Techniken 78
4. Sicherheit 79
5. Zusammenfassung 81

Literaturverzeichnis 83

Index 85

Einleitung

Die beständig steigenden Anforderungen von Anwendungen an die Leistungsmerkmale der sie ausführenden Rechnersysteme führen zu einem Bedarf an *Hochleistungsrechnern* (*high-performance computers*), deren Leistung über die herkömmlicher Einzelplatzrechner hinausgeht. Durch Vernetzung von Einzelplatzrechnern entsteht ein *Rechnerbündel* (*Cluster*) im *lokalen Netz* (*local area network*, LAN) bzw. *Rechnergitter* (*Grid*) im Falle eines Weitverkehrsnetzes (*wide area network, WAN*). Sie zeichnen sich typischerweise durch verteilte Ressourcen wie verteiltem Speicher (*distributed memory*) mit unterschiedlichen Zugriffszeiten (*non-uniform memory access*, *NUMA*) und eine Betriebssysteminstanz pro Rechnerknoten aus.

Die effiziente Parallelisierung von Anwendungen erfordert eine geeignete Zerteilung in möglichst unabhängige Teilaufgaben und die zeitliche und räumliche *Koordinierung* der Teilaufgaben wie auch der Betriebssysteminstanzen, etwa beim Zugriff auf gemeinsame Ressourcen, wie die folgenden zwei Beispiele zeigen sollen:

- **Lokaler Plattenspeicher.** Der durch die mechanische Konstruktion von Plattenspeichern bedingte nicht-wahlfreie Zugriff erfordert bei der gemeinsamen Nutzung von Plattenspeichern durch konkurrierende Prozesse besondere Maßnahmen zur Koordinierung der zeitlichen Zuteilung von Prozessen auf die Zugriffskanäle der Platte sowie bei der räumlichen Zuteilung der Daten auf der Platte. Die Ausarbeitung zum Thema *Disk Scheduling* befasst sich mit der zeitlich-räumlichen Zuteilung der Ressource Plattenspeicher zu auftraggebenden Prozessen.
- **Prozessoren.** Die quasi-parallele Abarbeitung der Aufgaben in Betriebssystemen durch *Prozesse* oder *Fäden* (*threads*), die auf Prozessoren ablaufen, erfordert eine Zuteilung von Prozessoren auf Prozesse. Im Falle paralleler Multiprozessorsysteme ist eine globale Koordinierung der Prozessorzuteilung wünschenswert, um im Sinne einer optimalen Prozessorauslastung auf allen Prozessoren stets mindestens einen Prozess laufbereit zu halten. Der Seminarbeitrag *Gang Scheduling* betrachtet Möglichkeiten und Grenzen der koordinierten Prozessorzuteilung in parallelen Systemen.

Die geographisch verstreuten Knoten eines Grid werden meist nicht von einer einzelnen Verantwortlichkeit betrieben und verwaltet. Daher sind *Selbstorganisation*, *Sicherheit* und *Fehlertoleranz* beim Grid von besonderer Bedeutung. Rechnergitter eignen sich wegen ihrer Weiträumigkeit insbesondere für solche verteilte Anwendungen, bei denen Kommunikation zwischen Knoten nur selten stattfindet. Auch beim Grid gibt es zahlreiche Koordinierungsaufgaben:

- **Netzwerkverbindungen.** Die räumliche Koordinierung bei der Zuteilung inhaltlich zusammenhängender Datenbestände auf topologisch nahegelegene Rechnerknoten wie im Falle *semantischer Überlagerungsnetzwerke* (*Overlay Networks*) will die Kommunikationslatenzen verringern helfen.

Ferner erfordern sich selbst organisierende Rechnerknoten eines Verbundes aus *gleichberechtigten Rechnerknoten* (*Peer-To-Peer-Netzwerke*) Mechanismen zur dezentralen Koordinierung global wirkender Operationen, wenn eine zentrale Instanz zur Koordinierung vermieden werden soll. Der Beitrag *Novel Network Systems* gibt einen Überblick über die aktuellen Trends im Bereich der Peer-To-Peer-Netze und Overlay-Techniken.

- **Netzwerkweite Speicherverbünde.** Sollen die lokalen Plattenspeicher der Rechnerknoten eines Rechnerverbundes organisatorisch zu einem netzwerkweiten Massenspeicherverbund zusammengefasst werden, so erfordert der effiziente Zugriff besondere Maßnahmen bei der Zuteilung der zu speichernden Daten auf die Rechnerknoten. Ferner ist eine Koordinierung beim Zugriff auf duplizierte Datenbestände (etwa bei Caching oder Replikation) zur Wahrung der Konsistenz bzw. Kohärenz notwendig. Der abschließende Beitrag *Internet-Wide Storage* untersucht verschiedene Ansätze, lokale Plattenspeicher von Rechnerknoten des Internet als virtuellen verteilten Massenspeicher zu nutzen.

KAPITEL 1

Disk Scheduling

Seminarbeitrag von **Lars Bauer**

1. Einleitung

Diese Ausarbeitung beschäftigt sich mit Disk-Scheduling. Dabei geht es darum, Festplattenanfragen von mehreren Anwendungsprogrammen möglichst schnell zu bearbeiten. Wichtig dafür ist eine je nach Situation geeignete Reihenfolge der Festplattenzugriffe festzulegen. Auch die Art und Weise, wie die Daten auf der Festsplatte abgelegt werden, spielt für das Erreichen einer hohen Verarbeitungsgeschwindigkeit eine große Rolle. Nach Vermittlung der zum Verständnis notwendigen Grundkenntnisse im folgenden Abschnitt werden anschließend drei moderne Disk-Scheduling-Verfahren genauer vorgestellt.

Die Prozessoren in den heutigen Rechnern sind in ihrer Leistungsfähigkeit auf einem sehr hohen Niveau. Auch die Leistungsfähigkeit der ersten Ebenen der Speicherhierarchie (Cache, RAM) und ihre Anbindung an die Prozessoren haben eine hohe Leistung erreicht. Dahingegen können die Festplatten, welche die Rolle des typischen Massendatenspeichers in den Computersystemen innehaben, nur geringe Verbesserungen bei der Zugriffszeit und der Übertragungsrate aufweisen. Dadurch ist die Lücke zwischen der Geschwindigkeit von Prozessoren und Arbeitsspeicher auf der einen Seite und den Festplatten auf der anderen Seite in den letzten Jahren merklich größer geworden. Umso wichtiger ist es, die eingeschränkte Leistung der heutigen Festplatten so gut wie möglich auszunutzen.

Die Betriebssysteme der heutigen Computer sind darauf ausgelegt, mehrere Anwendungsprogramme nebenläufig abzuarbeiten. Somit kommen die Festplattenzugriffsanforderungen im Allgemeinen von mehreren Anwendungen. Für diese Anforderungen muss eine Bearbeitungsreihenfolge gefunden werden. Diese Reihenfolge unterliegt je nach den Zielsetzungen des Computersystems gewissen Anforderungen, die sich in Extremfällen stark unterscheiden können. Das Festlegen dieser Reihenfolge der Festplattenzugriffe wird als Disk-Scheduling bezeichnet.

2. Grundlagen

In diesem Abschnitt werden die Grundlagen, die für das Verständnis der nachfolgenden Disk-Scheduling-Verfahren im dritten Abschnitt nötig sind, vorgestellt. Zuerst werden der typische Aufbau und die sich daraus ergebenden Zugriffsverzögerungen einer heutigen Festplatte erklärt. Im Anschluss wird eine Einführung in die

klassischen Disk-Scheduling-Verfahren gegeben, da die modernen Verfahren aus dem dritten Abschnitt teilweise auf diesen aufbauen. Für zusätzliche Grundlageninformationen sei auf weiterführende Literatur [**Sta98**], [**Pat03**] verwiesen.

2.1. Aufbau einer Festplatte. Aus der Sicht eines Anwendungsprogramms verhält sich eine Festplatte im wesentlichen wie ein lineares Stück Speicher, auf das wahlfrei zugegriffen werden kann. Für die Anwendungsprogramme ist diese Abstraktion ausreichend, für effiziente Disk-Scheduling-Verfahren werden jedoch detailliertere Kenntnisse über den Aufbau und die Funktionsweise einer modernen Festplatte benötigt.

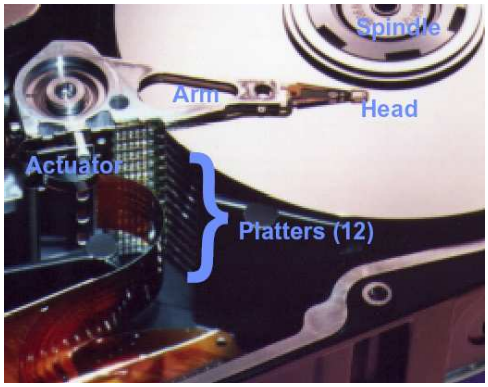


Abbildung 1. Aufbau einer Festplatte mit mehrere Platten und Schreib-/Leseköpfen

Eine Festplatte besteht für gewöhnlich aus mehreren magnetisierbaren Platten, die in Form eines Zylinders übereinander angeordnet sind. Für jede Platte ist ein Schreib-/Lesekopf vorhanden, der an einem beweglichen Arm befestigt ist. Im Betrieb schwebt der Schreib-/Lesekopf auf einem Luftpolster über der jeweiligen rotierenden Platte. Die Arme der Schreib-/Leseköpfe sind starr miteinander verbunden, wodurch sich alle Köpfe immer über der gleichen Stelle ihrer jeweiligen Platte befinden. Abbildung 1 zeigt ein Bild von dem Inneren einer Festplatte mit den einzelnen Magnetplatten und den dazugehörigen Schreib-/Leseköpfen. Bei einem Zugriff werden die Schreib-/Leseköpfe so ausgerichtet, dass die gewünschten Daten unter ihnen vorbeirotieren. An dieser Position verharren sie dann, bis die Daten vollständig gelesen oder geschrieben worden sind. Bei einem Schreibvorgang werden die Daten durch das gezielte Magnetisieren bestimmter Stellen auf den Platten abgelegt. Beim Lesen wird die Magnetisierung der Platten zu der ursprünglichen Information zurückinterpretiert.

Die einzelnen Platten sind unterteilt (Abbildung 2). Auf einer Platte befinden sich mehrere konzentrische Kreise, die **Spur** (Track) genannt werden. Diese Spuren sind wiederum in eine, pro Spur konstante Anzahl von **Sektoren** unterteilt. Moderne Festplatten gehen allerdings dazu über, in den größeren äußeren Spuren mehr Sektoren unterzubringen als in den inneren Spuren. Dazu wird die Anzahl der Spuren in mehrere Untergruppen eingeteilt, in denen die Anzahl der Sektoren pro Spur jeweils konstant ist. Diese Untergruppen werden Zonen oder Bänder genannt. Die Sektoren sind für die Festplattensteuerung elementare Einheiten. Das bedeutet, dass ein Schreib-/Leseauftrag des Schedulers immer eine Menge an Sektoren betrifft, die jeweils komplett bearbeitet werden. Allerdings wird nach außen von den eigentlichen Sektoren abstrahiert. Statt auf Sektoren wird der Scheduler auf **LBN**s (Logische Blocknummern) zugreifen, die festplattenintern eins zu eins auf Sektoren abgebildet werden. Dadurch kann bei Ausfall eines Sektors ein Reservesektor benutzt werden, ohne dass die LBN-Adressierung verändert werden muss. Da

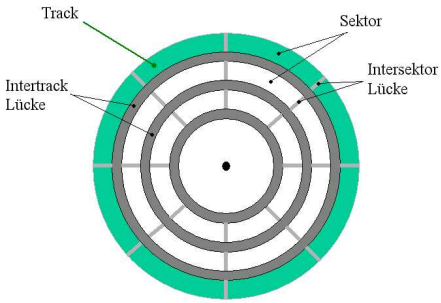


Abbildung 2. Die interne Aufteilung einer Platte

bis auf die Anordnung auf den Magnetplatten zwischen den Sektoren und den LBNs kein Unterschied besteht, werden beide Begriffe im Folgenden für eine elementare Dateneinheit auf der Festplatte stehen.

Zwischen benachbarten Sektoren und auch zwischen benachbarten Spuren befinden sich Lücken (Abbildung 2), in denen teilweise Verwaltungsdaten abgelegt sind. Diese Lücken dienen der besseren physikalischen Trennung der benachbarten Einheiten. Die Verwaltungsdaten werden unter anderem zur genaueren Positionierung des Schreib-/Lesekopfes genutzt.

2.2. Zugriffszeiten. Da einige der später vorgestellten Disk-Scheduling-Verfahren auf den genauen Ablauf eines Festplattenzugriffes aufbauen, soll dieser hier näher vorgestellt werden. Die Zugriffsaufforderung erfolgt durch den Disk-Scheduler, der nach der jeweils verwendeten Strategie einen Sektor der Festplatte als den nächsten zu bearbeitenden auswählt. Bevor ein spezieller Sektor der Festplatte bearbeitet werden kann, muss er sich unter dem Schreib-/Lesekopf befinden. Die Zeit, die bis zum Beginn der Bearbeitung vergeht, hängt im konkreten Fall von der momentanen Position des Schreib-/Lesekopfes, der Rotationsposition der Magnetplatte und den hardwarespezifischen Eigenschaften der jeweiligen Festplatte wie der Rotations- oder Kopfpositionierungsgeschwindigkeit ab.

Im Allgemeinen kann die Zeit, die für den Zugriff benötigt wird, in 3 Phasen unterteilt werden. Dabei wird davon ausgegangen, dass die Festplatte momentan nicht mit einem anderen Zugriff beschäftigt ist, der anstehende Zugriff durch den Disk-Scheduler also exklusiv auf der Festplatte durchgeführt werden kann. Als Erstes wird der Schreib-/Lesekopf zu der Spur bewegt, auf der sich der gesuchte Sektor befindet (**Suchzeit**). Als Nächstes muss gewartet werden, bis sich der gesuchte Sektor unter dem Kopf vorbeibewegt (**Rotationsverzögerung**). Die Summe aus der Suchzeit und der Rotationsverzögerung wird auch **Zugriffszeit** genannt. Wenn sich der gewünschte Sektor nun unter dem Kopf vorbeibewegt, kann in der dritten Phase die Schreib- oder Leseoperation durchgeführt werden (**Übertragungszeit**). Vor allem für die ersten beiden Phasen können nur Durchschnittswerte bestimmt werden, die je nach aktueller Kopfposition stark von der tatsächlich benötigten Zeit abweichen können.

Bei einigen Scheduling-Verfahren ist es nötig, sehr genaue Abschätzungen für die Zugriffszeit auf einen bestimmten Sektor zu besitzen, weil diese Information zum Beispiel zur Auswahl des nächsten zu bearbeitenden Auftrages herangezogen wird. Die Suchzeit, um den Kopf über der richtigen Spur zu positionieren, setzt sich dabei aus einer Startzeit, die benötigt wird, um den Kopf auf Geschwindigkeit zu bringen, und einer Bewegungszeit, die von der Anzahl der überquerten Spuren abhängig ist, zusammen. Die Rotationsverzögerung hängt hauptsächlich von der

Umdrehungsgeschwindigkeit und der momentanen Umdrehungsposition der Platte ab. Die anschließende Übertragungszeit wird ebenfalls stark von der Umdrehungsgeschwindigkeit der Platte mitbestimmt, aber auch die Datendichte, also die Anzahl der Bytes, die pro Spur untergebracht sind, spielt eine große Rolle.

2.3. Grundlegende Disk-Scheduling-Verfahren. Auf heutigen Computersystemen laufen üblicherweise Multitasking-Betriebssysteme. Das bedeutet, das mehrere Anwendungen nebenläufig, also quasi parallel ablaufen. Diese Anwendungsprogramme können jeweils Schreib- oder Leseaufträge absetzen. Durch die hohe Geschwindigkeit der heutigen Prozessoren im Vergleich zu den Bearbeitungszeiten der Festplatten können sich Plattenaufträge ansammeln. Da eine Festplatte ein exklusiv nutzbares Betriebsmittel ist, also die LBNs nur nacheinander bearbeitet werden können, müssen anstehende Aufträge in eine Reihenfolge gebracht werden, in der sie dann von der Festplatte bearbeitet werden sollen. Genau das ist die Aufgabe des Disk-Schedulers.

Die grundlegenden Verfahren zur Festlegung der Zugriffsreihenfolge basieren auf wohlbekannten Konzepten der Informatik. Die Warteschlange (**FIFO**) bearbeitet die Aufträge unabhängig von anderen Parametern in der Reihenfolge, in der sie beim Scheduler eingehen. Bei dem Konzept des Kellerspeichers (**LIFO**) wird der jeweils zuletzt eingegangene Auftrag zuerst bearbeitet. Der Vorteil der LIFO-Methode liegt darin, dass die Lokalität der Zugriffe erhöht wird, da der letzte eingegangene Auftrag mit einer hohen Wahrscheinlichkeit von dem gleichen Prozess kommt wie der momentan bearbeitete. Somit besteht die Möglichkeit, dass dieser zuletzt eingegangene Auftrag auf der Platte räumlich nahe dem momentan bearbeiteten Auftrag liegt und darum schnell von dem Schreib-/Lesekopf erreicht werden kann. Der Nachteil der LIFO-Methode ist es, dass ein Auftrag, der in dem Keller bereits vor längerer Zeit eingeordnet worden ist, eventuell gar nicht bearbeitet oder zumindest überdurchschnittlich lange verzögert wird, da neu eintreffende Aufträge immer bevorzugt behandelt werden.

Ein weiteres Verfahren, um das Problem der Zuteilung zu lösen, besteht darin, zufällig einen der Aufträge zur Bearbeitung auszuwählen. Beim Disk-Scheduling wird diese Methode jedoch nur für Analyse- und Simulationszwecke verwendet. Häufiger wird hingegen das **Prioritätenverfahren** eingesetzt, bei dem die eingegangenen Aufträge mit Prioritäten versehen sind und jeweils der Auftrag mit der höchsten Priorität ausgewählt wird. Die Prioritäten können dabei von den Anwendungsprogrammen oder dem Betriebssystem festgelegt werden. Weiterhin sind statische und dynamische Prioritäten unterscheidbar. Die Einstufung der dynamischen Prioritäten kann dabei im Gegensatz zu den statischen Prioritäten jederzeit verändert werden. Das Prioritätenverfahren entlastet den Scheduler, falls die Prioritäten von außen festgelegt werden. Alle bisher besprochen Scheduling-Verfahren lassen sich auf das Prioritätenverfahren reduzieren, wobei das Bestimmen der Prioritäten die Verhaltensweise des Verfahrens festlegt. So entspricht der Kellerspeicher zum Beispiel der Regel, dass neu eintreffende Anfragen die jeweils höchste Priorität erhalten.

2.4. Klassische Disk-Scheduling-Verfahren. Die vier in diesem Abschnitt vorgestellten klassischen Disk-Scheduling-Verfahren sind im Gegensatz zu den grundlegenden Verfahren aus dem vorangegangenen Abschnitt speziell für das Disk-Scheduling eingeführt worden. Bei dem Verfahren Shortest-Service-Time-First (**SSTF**) wird immer diejenige Anfrage als nächstes bearbeitet, bei welcher der Arm am wenigsten bewegt werden muss. Es wird also immer die kürzeste Suchzeit bevorzugt. Dadurch kann es passieren, dass sich der Arm durch neu eingehende Aufträge nur noch lokal in einer bestimmten Gegend der Platte bewegen wird und Anfragen auf einer weiter entfernten Spur nur mit starker Verzögerung oder im

Grenzfall gar nicht bearbeitet werden. Darum werden in der Praxis nur modifizierte SSTF-Verfahren eingesetzt. Obwohl durch SSTF immer der Auftrag mit der lokal geringsten Suchzeit zuerst bearbeitet wird, kann nicht die global kürzeste Suchzeit garantiert werden, da nach einer Folge von schnellen lokalen Zugriffen einige weiter entfernte Zugriffe die Durchschnittszeit stark verschlechtern können.

Abgesehen von dem FIFO-Konzept haben alle bisher vorgestellten Verfahren die Gemeinsamkeit, dass unter ungünstigen Bedingungen einige Aufträge niemals bearbeitet werden. Ein Verfahren, das ebenfalls garantiert, dass alle eingegangene Aufträge tatsächlich bearbeitet werden, ist die Fahrstuhlstrategie (**SCAN**). Dabei bewegt sich der Arm mit dem Schreib-/Lesekopf zuerst nur in eine Richtung, also zum Beispiel vom Inneren der Platte zum Äußeren, und bearbeitet auf diesem Weg alle Aufträge, deren Zugriffsposition passiert werden. Wenn der Arm den Rand der Platte erreicht oder keine weiteren Aufträge mehr in der momentanen Bewegungsrichtung vorhanden sind, wird die Bewegungsrichtung umgekehrt und nach dem gleichen Verfahren alle Aufträge auf dem Rückweg bearbeitet.

Durch das SCAN-Verfahren werden die Aufträge im Allgemeinen nicht gleichmäßig fair behandelt. So werden die Aufträge, deren Zugriffsposition am äußeren oder inneren Rand der Platte liegen, teilweise bevorzugt. Wenn der Kopf beispielsweise in der einen Richtung an den äußeren Spuren vorbeibewegt wurde und dabei Aufträge bearbeitet hat, können Folgeaufträge, die lokal in derselben Gegend liegen, mit einer kurzen Wartezeit bei der folgenden Rückwärtsbewegung des Kopfes abgearbeitet werden. Es kann aber auch zu Benachteiligungen eben dieser äußeren und inneren Spuren kommen, wenn zum Beispiel ein Auftrag am äußeren Rand der Platte erst dann eintrifft, nachdem der Kopf bereits in der Richtung zum anderen Ende der Platte an der Zugriffsposition dieses neuen Auftrages vorbeibewegt wurde. Dann wird dieser Auftrag erst bearbeitet, wenn der Kopf die gegenüberliegende Spur erreicht und sich wieder zurückbewegt hat.

Eine Möglichkeit, um diese Ungleichmäßigkeiten bei der SCAN-Methode zu beheben, ist das Verfahren **C-SCAN**, bei dem die Bearbeitung der Aufträge nur in einer Bewegungsrichtung des Kopfes durchgeführt wird. Wenn der Rand der Platte erreicht ist oder keine weiteren Aufträge mehr in der Bewegungsrichtung vorliegen, bewegt sich der Kopf in einem Zug bis zum anderen Ende der Platte zurück, ohne dabei eventuell vorhandene Aufträge zu bearbeiten. Von dort beginnt er dann in der gleichen Richtung wie beim ersten Durchgang mit den neuen Aufträgen.

Auch bei SCAN und C-SCAN kann es vorkommen, dass der Kopf durch ständig neu eintreffende Aufträge auf derselben oder nachfolgenden Spuren für lange Zeit in derselben Gegend verharrt. Um dieses Problem abzuschwächen wurden die Varianten **N-Step-SCAN** und **FSCAN** eingeführt. Beiden ist gemeinsam, dass mehrere Warteschlangen für neu eingehende Aufträge vorhanden sind. Während die Aufträge einer Warteschlange über eine komplette Armbewegung abgearbeitet werden, kommen neu eingehende Aufträge in eine andere Warteschlange und können somit den Kopf nicht an einer Stelle festhalten. Der Unterschied zwischen den beiden Verfahren liegt in der Anzahl und Größe der zusätzlichen Warteschlangen. Bei N-Step-SCAN sind beliebig viele Warteschlangen mit der maximalen Größe N vorhanden. Bei einem Durchgang werden höchstens die ersten N Aufträge abgearbeitet, während neu eingehende Aufträge in die folgenden Warteschlangen eingefügt werden. Dahingegen verfügt FSCAN über genau zwei Warteschlagen beliebiger Größe. Während die eine Schlage abgearbeitet wird, werden neu eintreffende Aufträge in die andere Schlange eingeordnet. Nachdem alle Aufträge der aktuellen Warteschlange abgearbeitet sind, ändert der Kopf die Bewegungsrichtung und bearbeitet die Aufträge der anderen Schlange.

3. Aktuelle Disk-Scheduling-Verfahren

Die drei Disk-Scheduling-Verfahren, die in diesem Abschnitt vorgestellt werden, entstammen drei unterschiedlichen Arbeiten ([**ID01**], [**SGLG02**], [**LSG02**]). Als erstes wird das Anticipatory Scheduling vorgestellt, das die sequentiellen Zugriffe von synchronen Programmen beschleunigt. Danach werden die Track-Aligned Extents besprochen, welche durch eine geschickte Ablage der Daten auf der Platte die Zugriffszeit verbessern. Als drittes Verfahren wird das Freeblock-Scheduling erläutert, bei dem die Verzögerungszeiten zur Bearbeitung von Hintergrundaufträgen genutzt werden.

3.1. Anticipatory Scheduling. Anticipatory Scheduling versucht das Problem des Nichterkennens von sequentiellen Zugriffen, die durch eine kurze Berechnungsphase unterbrochen sind, zu lösen. Der Ansatz besteht darin, den Scheduler nach der Erfüllung einer Zugriffsanforderung kurz warten zu lassen, um das Ende der Berechnungsphase abzuwarten und so die Möglichkeit zu erhalten, den sequentiellen Folgezugriff zu bearbeiten.

Motivation und Funktionsweise. Das englische Wort Anticipatory bedeutet vorausahnen. Damit wird eine wichtige Grundeigenschaft dieses Scheduling-Verfahrens beschrieben, denn das Verfahren benutzt eine Beobachtung aus der Vergangenheit, um eine Vorhersage über zukünftige Verhaltensmuster zu treffen. Diese Vorhersage wird genutzt, um das Scheduling-Verfahren für das erwartete Verhaltensmuster zu optimieren.

Beobachtet wird das Verhalten von typischen Anwendungsprogrammen in Bezug auf ihre Festplattenaktivitäten. Es kommt dort nur sehr selten vor, dass von einem Anwendungsprogramm ein einzelner Sektor bearbeitet wird. Üblicherweise wird eine gewisse Menge an räumlich auf der Platte benachbarten Sektoren bearbeitet. Trotzdem bekommt das Scheduling-Verfahren von dem Anwendungsprogramm nicht einen einzigen großen Auftrag, sondern nacheinander eine Menge von kleinen, zusammenhängenden Aufträgen. Das typische Ablaufmuster in einem exemplarischen Anwendungsprogramm, auf das im weiteren Bezug genommen wird, sieht dabei so aus, dass zuerst eine kleine Datenmenge von der Festplatte geladen und anschließend verarbeitet wird. Im Anschluss an die Verarbeitung werden die nächsten Daten beim Scheduler angefordert und nach der Lieferung ebenfalls verarbeitet. Dieses Muster wiederholt sich, bis die gesamte Datenmenge abgearbeitet ist.

Der Nachteil dieses typischen Zugriffsmusters ist, dass es der Arbeitsweise der klassischen Scheduling-Verfahren Probleme bereitet. Die klassischen Scheduling-Verfahren sind so ausgelegt, dass sie nach Möglichkeit immer beschäftigt sind, also insbesondere keine Leerlaufzeiten haben, um eine insgesamt hohe Auslastung zu erreichen. Das bedeutet für das vorgestellte Anwendungsprogramm, dass der Scheduler unmittelbar nach der Erfüllung des ersten Zugriffswunsches damit beginnt, einen anderen Auftrag aus seiner Auftragsliste zu bearbeiten. Dafür muss der Schreib-/Lesekopf im Allgemeinen an eine andere Position bewegt werden, bevor der erneute Zugriff auf die Nutzdaten beginnen kann.

Bei der Geschwindigkeit heutiger Mikroprozessoren ist der Zeitraum zwischen dem Erhalten der zuletzt angeforderten Festplattendaten des exemplarischen Anwendungsprogramms und dem Anfordern der nächsten Festplattendaten desselben Programms üblicherweise sehr kurz. Trotzdem hat der Scheduler in der Zwischenzeit möglicherweise bereits damit begonnen, den Schreib-/Lesekopf von der aktuellen Position wegzubewegen, um den Auftrag eines anderen Programms zu bearbeiten. Je nach Verarbeitungsgeschwindigkeit des Mikroprozessors und der Komplexität der Datenverarbeitung des exemplarischen Anwendungsprogramm ist die Abarbeitung des Festplattenzugriffswunsches von dem anderen Programm noch nicht sehr weit fortgeschritten, wenn die Folgeanfrage des exemplarischen Anwendungsprogramms

abgesetzt wird. Da die Festplatte jedoch bereits mit der Bearbeitung des anderen Zugriffswunsches begonnen hat, benötigt es einen längeren Zeitraum, bis die Folgeanfrage des exemplarischen Anwendungsprogramms bearbeitet werden kann.

Hätte der Scheduler stattdessen einen kurzen Augenblick gewartet, bevor er mit der Bearbeitung eines anderen Auftrages beginnt, hätte das Anwendungsprogramm die Chance gehabt, die gerade erhaltenen Daten zu bearbeiten und dann im Anschluss den nachfolgenden Zugriffswunsch abzusetzen. Dieser neue Zugriffswunsch befindet sich dabei oftmals in der unmittelbaren Nähe des letzten Zugriffes, so dass der Schreib-/Lesekopf dafür in vielen Fällen nicht neu positioniert werden muss. Je nach verwendeter Scheduling-Strategie hätte der Scheduler dann die Möglichkeit gehabt, gerade diesen neu eingetroffenen Zugriffswunsch als den nächsten zu bearbeitenden auszuwählen und somit eine aufwändige Repositionierung des Kopfes zu vermeiden. Der Zeitverlust, den das Warten auf einen eventuellen Folgezugriffswunsch verursacht, wird durch die Möglichkeit, diesen neuen Auftrag besonders schnell zu bearbeiten, ausgeglichen.

Beispiele. Abgesehen von der Möglichkeit, einen insgesamt höheren Datendurchsatz zu erreichen, kann der Ansatz des Anticipatory Scheduling dem ursprünglichen Scheduling-Verfahren auch zum Einhalten seiner primären Ziele verhelfen. Bei einem Computersystem, das als Scheduling-Verfahren zur Minimierung der Suchzeit SSTF einsetzt, kann es passieren, dass dieses Scheduling-Verfahren zu dem Verhalten einer Warteschlage degeneriert, wodurch die Systemperformance merklich sinkt. In diesem Computersystem sollen nun exemplarisch zwei Prozesse laufen, die jeweils eine große Datei einlesen. Das erwartete Verhalten für einen die Suchzeit minimierenden Scheduler wäre, dass zuerst einige Anfragen des einen Prozesses bearbeitet werden, um danach eine teure Suchoperation zu der Plattenposition der anderen Datei durchzuführen, wo dann einige Anfragen des anderen Prozesses bearbeitet werden. Durch das typische Verhalten der Anwendungen, zwischen zwei Anfragen eine kurze Berechnungsphase einzulegen, hat der Scheduler zu der Zeit, bei der die Entscheidung getroffen werden muss, welcher Auftrag als nächstes bearbeitet werden soll, keine Auswahl. Es steht nur die Anfrage des jeweils anderen Anwendungsprogramms zur Verfügung, so dass er nach dem Bearbeiten von nur einer Anfrage des ersten Anwendungsprogramms eine teure Suchoperation zu der Plattenposition der Datei des anderen Programms machen muss. Nach der Abarbeitung dieser Anfrage steht das Scheduling-Verfahren erneut vor dem Problem und muss mit einer erneuten Suchoperation zu der Plattenposition des ersten Programms zurückwechseln. Durch diese Degeneration auf ein FIFO Verhalten geht ein großer Teil der verfügbaren Übertragungsbandbreite verloren.

Eine andere Klasse von Scheduling-Verfahren, die durch die typische Abfolge von Daten Holen und Bearbeiten nicht mehr in der Lage ist, ihre primären Ziele einzuhalten, ist die Klasse der Proportional-Share-Verfahren. Diese ermöglichen es dem Anwender genau festzulegen, wie das Bearbeitungsverhältnis von mehreren Anwendungsprogrammen sein soll. So kann beispielsweise eingestellt werden, dass ein Anwendungsprogramm die dreifache Festplattenbandbreite erhalten soll wie ein anderes. Wenn die beiden Programme ungefähr gleich viele Anfragen mit Berechnungsphasen zwischen den Anfragen stellen, so ist das Scheduling-Verfahren jedoch nicht in der Lage, diese Vorgabe einzuhalten. Der Grund dafür ist wieder das sofortige Wechseln zu dem jeweils anderen Anwendungsprogramm nach der Abarbeitung eines Auftrags, da keine Bearbeitungsalternative vorhanden ist.

Diese Probleme treten in beiden Beispielen auch dann auf, wenn mehr als nur zwei Anwendungsprogramme um die Festplatte konkurrieren. Mit der Erweiterung durch das Anticipatory Scheduling können die Probleme in beiden Fällen gelöst werden. Durch das kurze Warten erhält das Scheduling-Verfahren einen zusätzlichen Auftrag, der ausgewählt werden kann. Häufig ist es gerade dieser Auftrag, der es

dem zu Grunde liegenden Scheduling-Verfahren ermöglicht, sein eigentliches Ziel zu erfüllen.

Implementierung. Die Frage, ob und wie lange der Scheduler warten soll, bis er sich für den nächsten zu bearbeitenden Auftrag entscheidet, ist die Schlüsselstelle für eine effiziente Implementierung. Dabei hängt die Antwort auf diese Frage zu einem großen Teil auch von dem zu Grunde liegenden Scheduling-Verfahren ab. Darum wurde in der gewählten Implementierung von der Beantwortung dieser Frage abstrahiert. Abbildung 3 zeigt die drei Hauptkomponenten der Implementierung. Ganz außen befindet sich die Kernimplementierung des Anticipatory-Verfahrens als Schnittstelle für die Anwendungsprogramme. Sie kapselt das zugrundeliegende Verfahren, das zum Beispiel eine Variante von SSTF oder CSCAN sein kann. Passend zu diesem zu Grunde liegenden Verfahren muss eine Heuristik bereitgestellt werden, die von dem Anticipatory-Verfahren zur Festlegung der Wartezeit für den konkreten Fall benutzt wird.

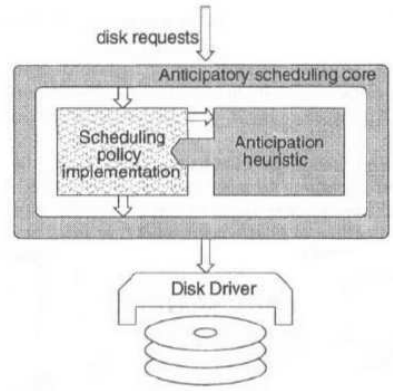


Abbildung 3. Rahmenwerk des Anticipatory Schedulings

Ein Vorteil dieses Rahmenwerkes ist es, dass die Anpassung des Rahmens an ein neues Scheduling-Verfahren sehr einfach ist. Es muss lediglich eine neue Heuristik gefunden werden, die sich für die Ziele des Scheduling-Verfahrens eignet. Durch das sehr allgemein gehaltene Rahmenwerk, das den zugrunde liegenden Scheduler unverändert in das neue Gesamtverfahren übernimmt, kann das Anticipatory-Verfahren auch leicht in bestehende Betriebssysteme integriert werden. Für die durchgeführten Testfälle wurde das Anticipatory Scheduling in den FreeBSD-4.3-Kern integriert.

Ergebnisse. Die Ergebnisse der durchgeführten Messungen zeigen, dass grundsätzlich eine Verbesserung des Durchsatzes erreicht werden kann. Teilweise ist der Zuwachs sehr gering, wenn zum Beispiel ein Mechanismus im Betriebssystemkern einen sequentiellen Dateizugriff erkennt und daraufhin selbstständig weitere Lesezugriffe auf Verdacht absetzt. In den Fällen, in denen dieser sequentielle Zugriff nicht erkannt wird, wird eine Erhöhung des Durchsatzes um bis zu einem Faktor vier erreicht. Im Durchschnitt wird bei Anwendungsprogrammen, die nach dem typischen Laden-Berechnen-Muster agieren, eine Verdoppelung des Durchsatzes erreicht.

Es gibt weitere Möglichkeiten, um dem beschriebenem Problem zu begegnen. Im wesentlichen beruhen diese Alternativen darauf, zusätzliche Aufträge an den Scheduler frühzeitig abzusetzen, um so das Problem, dass der Scheduler zur Zeit der Entscheidung noch nicht alle relevanten Zugriffswünsche kennt, zu umgehen. Das Problem bei diesen Alternativen liegt darin, diese zusätzlichen Aufträge frühzeitig zu bestimmen. Wenn das Anwendungsprogramm selbst weiß, welche Anfragen es in

naher Zukunft stellen wird, dann könnten diese als asynchrone Aufträge abgeschickt werden. Diese asynchronen Aufträge blockieren das Anwendungsprogramm nicht, so dass es zunächst weiterarbeiten kann, bis die jeweiligen Daten der asynchronen Zugriffe tatsächlich benötigt werden. Der Nachteil dieser Lösung liegt darin, dass zum einen die bestehenden Anwendungsprogramme dahingehend umgeschrieben werden müssten und zum anderen die Möglichkeit der asynchronen Zugriffe nicht in jedem Betriebssystem vorgesehen ist. Außerdem gibt es viele Anwendungsprogramme, die keine Kenntnis über die eigenen Zugriffswünsche in naher Zukunft haben, da diese zum Beispiel stark von Benutzereingaben abhängen.

Eine transparentere Möglichkeit sind die präventiven Zugriffe, die vom Betriebssystemkern abgesetzt werden. Hier versucht das Betriebssystem selbst, das Zugriffsmuster der Anwendungsprogramme zu erkennen. Diese Vorgehensweise wird in der Praxis häufig eingesetzt, jedoch haben die Betriebssysteme in aller Regel noch weniger Kenntnis von zukünftigen Zugriffen der Anwendungsprogramme als diese selbst. Dadurch wird die Vorhersage der Zugriffsposition extrem kompliziert, und auch die Kosten bei einer falschen Vorhersage sind im Allgemeinen sehr hoch, da die spekulativ abgesetzten Zugriffe auf jeden Fall abgearbeitet werden, gleichgültig, ob sie letztendlich benötigt werden oder nicht.

Insgesamt haben diese Verfahren gegenüber dem Anticipatory-Scheduling nur den Vorteil, auf die kurze Wartezeit vor der Auswahl des nächsten zu bearbeitenden Auftrages verzichten zu können. Die aufgezählten Nachteile der Alternativen überwiegen diese kurze Pause jedoch deutlich, vor allem, da die Pause durch zukünftige schnellere Prozessoren weiter verringert werden kann.

3.2. Track-Aligned-Extents. Ein häufig auftretendes Problem bei einem Festplattenzugriff ist es, dass die angeforderten Daten, über eine Spurgrenze hinweg abgelegt sind, wodurch der Zugriff merklich verzögert wird. Durch das Ausrichten von zusammenhängenden Daten an diesen Spurgrenzen beschleunigen die Track-Aligned-Extents den Zugriff.

Motivation und Funktionsweise. Das Verfahren der Track-Aligned-Extents ist kein Scheduling-Verfahren und nimmt auch nicht Bezug auf ein spezielles Scheduling-Verfahren. Stattdessen soll die Leistungsfähigkeit der Festplatte durch ein geschicktes Platzieren der Nutzdaten erhöht werden. Dafür ist eine detaillierte Kenntnis des Aufbaus einer modernen Festplatte nötig. Wieder ist eine Beobachtung aus der Praxis die Grundlage für diese Verbesserung.

Wenn auch nicht in dem Maße, wie bei anderen Teilen eines heutigen Computers, wie zum Beispiel dem Prozessor, so sind doch auch bei den Festplatten im Laufe der Jahre kontinuierlich Verbesserungen erzielt worden, mit denen die Geschwindigkeit und die Kapazität erhöht werden konnten. Eine Phase beim Datenzugriff konnte jedoch nur unwesentlich beschleunigt werden und es scheint auch in der Zukunft keine relevante Verbesserung erreichbar zu sein. Es handelt sich um die Positionierung des Schreib-/Lesekopfes, die in der Zukunft eher länger dauern wird, da durch die ständig vergrößerte Datendichte die Positionierung auf der Seite der Feinmechanik immer komplizierter wird. So dauert der Wechsel von einer Spur auf die benachbarte bei einer aktuellen Festplatte zwischen 0,6 und 1,1 ms. Das entspricht in etwa 20% der Zeit, die für eine vollständige Umdrehung der Platte bei 15.000 U/min benötigt wird oder etwa 800.000 Takten eines 1 GHz Prozessors.

Im Durchschnitt wären bei der Rotationsverzögerung eine halbe Umdrehung der Platte erwartbar, so dass die 20% der Plattenumdrehung, die von der Kopfpositionierung benötigt werden, nicht maßgeblich ins Gewicht fallen. Durch die Zero-Latency-Access-Technik, die in der Firmware von modernen Festplatten implementiert ist, fällt die durchschnittliche Rotationsverzögerung jedoch wesentlich geringer aus, so dass die Kopfpositionierungszeit den dominanten Anteil bilden kann. Ohne

die Zero-Latency-Access-Technik wird bei einem Lesezugriff auf mehrere aufeinanderfolgende Sektoren in einer Spur nach der Kopfpositionierung üblicherweise gewartet, bis der erste Sektor des Auftrages unter dem Schreib-/Lesekopf vorbeirotiert, um dann ihn und anschließend die folgenden Sektoren einzulesen. Durch die Zero-Latency-Access Technik muss nicht mehr gewartet werden, bis der erste Sektor den Kopf passiert. Stattdessen können die angeforderten Sektoren in jeder beliebigen Reihenfolge eingelesen werden. Sie werden dann festplattenintern umsortiert und anschließend in der richtigen Reihenfolge ausgeliefert. Dadurch kann die Rotationsverzögerung in vielen Fällen deutlich verringert werden.

Durch die hohen Kosten einer Kopfneupositionierung sollte ein Spurwechsel innerhalb eines Auftrages nach Möglichkeit vermieden werden, da ein Zugriff auf eine Menge von aufeinanderfolgenden und komplett innerhalb einer Spur liegenden Sektoren bedeutend schneller geht als ein Zugriff auf die gleiche Anzahl an Sektoren, die jedoch auf zwei benachbarte Spuren verteilt sind. Die einzige Möglichkeit, einen solchen Spurwechsel zu vermeiden ohne die Aufträge der Anwendungsprogramme zu ändern, besteht darin, bereits beim Zuteilen von Speicherplatz auf die Spurgrenzen zu achten. Dadurch können zusammenhängende Daten in vielen Fällen ohne Überschreitung einer Spurgrenze abgespeichert werden. Dafür müssen diese Spurgrenzen jedoch zuerst bekannt sein. Im Idealfall könnten die Informationen über Spurgrenzen direkt bei den Festplatten erfragt werden. Da die Festplatten jedoch von diesen Details abstrahieren, um einen einfacheren und einheitlicheren Zugriff zu ermöglichen, müssen die Grenzen zuerst durch eine Untersuchung ermittelt werden.

Implementierung. Für die Implementierung muss zuerst ein allgemeingültiger Weg gefunden werden, wie die Spurgrenzen bestimmt werden können. Aus Sicht der Anwendungsprogramme verhält sich die Festplatte wie eine lineare Abfolge von elementaren Speichereinheiten. Diese durchnummerierten Speichereinheiten werden über LBNs adressiert. Die Festplatte bildet intern jede LBN auf einen bestimmten Sektor einer bestimmten Spur ab. Die Frage ist also, zwischen welchen LBNs die Spur gewechselt wird. Diese Grenzen zu bestimmen erweist sich als sehr aufwändig. Das liegt vor allem daran, dass aufeinanderfolgende LBNs nicht unbedingt aufeinanderfolgenden Sektoren zugeordnet werden.

Die Festplattenhersteller erwarten, dass einige Sektoren aufgrund von Fertigungsungenauigkeiten nicht richtig funktionieren. Diese Defekte können entweder bereits vor der Auslieferung, oder aber erst nach einer Weile der Benutzung auftreten. Darum sind Reservesektoren vorgesehen, um die versprochene Gesamtkapazität auch in solchen Fällen erreichen zu können. An welchen Stellen der Platte sich diese Reservesektoren befinden, ist sehr unterschiedlich. Es gibt Festplatten, bei denen jede Spur ihre eigenen Reservesektoren hat, es kommt aber auch vor, dass die Reservesektoren gesammelt am Ende der Platte zusammengefasst sind. Durch das Auftreten von Sektorfehlern sind auch innerhalb der verschiedenen Zonen einer Platte, die eigentlich eine jeweils konstante Anzahl an Sektoren pro Spur haben sollten, Abweichungen der Sektoranzahl möglich.

Besonders problematisch wird es, wenn während des Betriebs ein Sektor ausfällt und durch einen Reservesektor ersetzt werden muss. Auch für die Lösung dieses Problems gibt es verschiedene Ansätze. Das beim Auftreten des Fehlers Einfachste ist es, diesen Sektor als defekt zu markieren und stattdessen einen der Reservesektoren für die betroffene LBN zu verwenden. Dadurch gäbe es aber einen Sprung in der Abbildungsvorschrift, welche den Sektoren die LBNs zuteilt und es gäbe bei einem sequentiellen Zugriff eventuell auch zusätzliche Spurwechsel, falls der Reservesektor nicht auf der gleichen Spur liegt wie der ursprüngliche Sektor. Darum wird stattdessen üblicherweise der defekte Sektor übersprungen und die nachfolgenden Sektoren werden in Bezug auf ihre LBN Zuteilung neu durchnummeriert, bis ein Reservesektor erreicht wird, der den übriggebliebenen Sektor aufnimmt. Das ist zwar in dem

Moment aufwändiger, weil Datenblöcke verschoben werden müssen, es hilft jedoch, in der Zukunft effizienter arbeiten zu können.

Bis ein neuer Sektor ausfällt, was für gewöhnlich nur sehr selten passiert, ändern sich die Spurgrenzen auf der Platte nicht. Diese müssen also meistens nur ein einziges Mal bestimmt werden. Dafür wurde ein sehr allgemeiner Ansatz entwickelt, der nur lesend auf die Festplatte zugreifen muss. Die Kernidee besteht darin, die Verzögerung, die ein Spurwechsel mit sich bringt, durch Zeitmessungen zu erkennen. Dafür werden, bei einer bestimmten LBN L beginnend, in mehreren Durchläufen eine jeweils um eins vergrößerte Anzahl an LBNs gelesen: im ersten Durchlauf also nur die LBN L, danach L und L+1, anschließend L bis L+2, und so weiter. Solange diese LBNs auf derselben Spur liegen, kann mit einem ungefähr linearen Anwachsen der Lesezeit gerechnet werden. Sobald aber die Spurgrenze überschritten wird, steigt die benötigte Zeit für die gesamte Leseoperation deutlich an. Durch diesen Unterschied in der Lesezeit kann die Spurgrenze ermittelt werden. Sobald eine Spurgrenze bestimmt wurde, beginnt das Verfahren mit dem ersten LBN der neuen Spur von vorne.

Um die Messungen so genau durchführen zu können, wie es zur Bestimmung der Spurgrenzen nötig ist, müssen störende Faktoren wie zum Beispiel die Rotationsverzögerung vor dem Beginnen des eigentlichen Lesevorgang soweit wie möglich eliminiert werden. Dafür wird das Abschicken des Leseauftrags an die Festplatte mit der Rotation der Festplatte synchronisiert, so dass jeder Leseauftrag ungefähr an der gleichen Rotationsposition ankommt. Auch die Cacheeffekte müssen berücksichtigt werden, da beim Auffinden einer Spurgrenze bis auf die jeweils neu hinzukommende LBN immer wieder die gleichen Stellen der Platte gelesen werden. Um die untersuchten Sektoren aus dem Cache zu verdrängen, werden 100 weit über die Platte verstreute Leseoperationen zwischen den eigentlichen Zugriffen eingefügt.

Auf einer typischen 9GB großen Platte dauert der ganze Vorgang 4 Stunden. Abgesehen von diesem allgemeinen Ansatz wurde aber auch noch ein speziell für das SCSI-Protokoll vorgesehener Ansatz entwickelt, der hier jedoch nicht weiter vorgestellt werden soll. Durch das geschickte Ausnutzen des komplexeren SCSI-Befehlssatzes konnte der gesamte Vorgang auf einer gleich großen Platte in 5 Minuten erledigt werden.

Ergebnisse. Messungen auf einem um das Verfahren der Track-Aligned-Extents erweiterten FreeBSD System ergeban, dass sich hauptsächlich bei Dateien mittlerer Größe (100-500KB) Verbesserungen feststellen lassen. Das liegt daran, dass kleine Dateien auch ohne spezielle Vorsorge nur sehr selten eine Spurgrenze überschreiten und besonders große Dateien für gewöhnlich gleich mehrere Spurgrenzen überschreiten, so dass die Anzahl der Spurüberschreitungen bei ihnen durch gezieltes Ausrichten höchstens um eine verringert werden kann.

Für Aufträge mittlerer Größe wird die Platteneffizienz um bis zu 50 Prozent gesteigert. Die Platteneffizienz ist dabei der Anteil an der gesamten Bearbeitungszeit, bei der die tatsächlichen Nutzdaten bearbeitet werden, also ohne Berücksichtigung der mechanisch bedingten Verzögerungszeiten. Darüber hinaus wird eine signifikante Verringerung der Standardabweichung der Antwortzeit erreicht. Die Verringerung ist umso größer, je näher die Auftragsgröße an der Spurgröße liegt. Das ist besonders wichtig, um die Antwortzeiten präzise vorhersagen zu können. Echtzeitanwendungen erhalten dadurch genauere Schranken für ihre Abschätzungen. Auch die deutliche Verbesserung der schlechtmöglichsten Zugriffszeit kommt den Echtzeitanwendungen zu gute.

3.3. Freeblock-Scheduling. Ein Problem bei einem Festplattenzugriff ist es, dass ein Großteil der gesamten Bearbeitungszeit gewartet werden muss, bis die

angeforderten Daten unter dem Schreib-/Lesekopf vorbeirotieren. Das Freeblock-Scheduling nutzt diese Rotationsverzögerung, um andere Aufträge zu bearbeiten, die an geeigneten Stellen liegen.

Motivation und Funktionsweise. Das Freeblock-Scheduling versucht die Untätigkeit während der Rotationsverzögerung durch das Bearbeiten von anderen Aufgaben geschickt auszunutzen. Zwar wurde im vorangegangenen Abschnitt gezeigt, dass durch die Technik des Zero-Latency-Access diese Rotationsverzögerung teilweise deutlich verringert werden kann, aber diese Technik bringt nicht für jeden Auftrag eine Verbesserung, so dass die Rotationsverzögerung im Allgemeinen immer noch einen guten Ansatzpunkt für Verbesserungen bildet.

Nachdem ein Auftrag bearbeitet und anschließend ein neuer Auftrag ausgewählt worden ist, dauert es einige Zeit, bis sich der Schreib-/Lesekopf über dem richtigen Sektor befindet. Zuerst muss der Schreib-/Lesekopf zur richtigen Spur bewegt werden, anschließend wird gewartet, bis die richtige Stelle dieser Spur unter dem Schreib-/Lesekopf vorbeirotiert. Gerade dieses Warten kann durch die Bearbeitung von Hintergrundaufträgen überbrückt werden, ohne dabei die Bearbeitungszeit der Vordergrundaufträge zu erhöhen. Diese Hintergrundaufträge müssen dabei jeweils so auf der Platte gelegen sein, dass sie während der Rotationsverzögerung bearbeitbar sind.

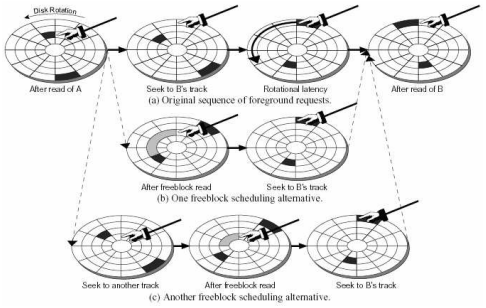


Abbildung 4. Zwei beispielhafte Freeblock-Scheduling Möglichkeiten

Abbildung 4 zeigt zwei beispielhafte Aufträge, die jeweils in der Rotationsverzögerung der eigentlichen Hauptaufträge bearbeitet werden können. In Teil (a) werden nur diese beiden Hauptaufträge gezeigt. Der auf der inneren Spur gelegene schwarze Kasten markiert dabei den Auftrag A, der auf der äußeren Spur den Auftrag B und die stilisierte Hand steht für den Schreib-/Lesekopf. Im dritten Bild von Teil (a) wird die Rotationsverzögerung verdeutlicht, bei welcher herkömmliche Systeme warten. In Teil (b) werden stattdessen direkt nach der Bearbeitung von Auftrag A weitere Sektoren auf der gleichen Spur, auf der auch Auftrag A platziert war, eingelesen. Wichtig dabei ist, dass nur so viele Sektoren eingelesen werden, dass der Wechsel zur Spur von Auftrag B noch rechtzeitig durchgeführt werden kann, ohne eine zusätzliche Umdrehung abwarten zu müssen. Diese Sektoren sind nun eingelesen worden, ohne die Bearbeitungszeit der Aufträge A oder B zu beeinflussen. Das Gleiche gelingt auch in Teil (c), bei dem diese zusätzlichen Sektoren jedoch auf einer anderen Spur gelegen sind, zu dem nach Beendigung des Auftrages A zuerst gewechselt werden muss. Durch diesen zusätzlichen Wechsel können insgesamt weniger Sektoren eingelesen werden als in Teil (b), aber trotzdem kann sich dieser zusätzliche Wechsel lohnen.

Das Konzept des Freeblock-Scheduling beeinflusst nicht den eigentlichen Scheduler. Der ursprünglich vom Betriebssystem vorgesehene Scheduler kann weiterhin seine Arbeit tun und er wird dabei von den Freeblock-Erweiterungen nicht gestört. Zusätzlich zu dem ursprünglichen Scheduler gibt es jetzt einen Freeblock-Scheduler,

der aus einer eigenen Auftragsliste Aufträge auswählt und sie so zwischen den Aufträgen des eigentlichen Schedulers platziert, dass dessen Aufträge dadurch nicht verzögert werden. Diese Freeblock-Aufträge werden also zusätzlich zu den ursprünglichen Aufträgen ausgeführt.

Nun müssen noch Aufträge gefunden werden, die sich als Freeblock-Aufträge eignen. Charakteristisch für Freeblock-Aufträge ist, dass sie eine sehr niedrige Ausführungspriorität haben und eine große Menge an Sektoren bearbeiten, die außerdem keine besondere Bearbeitungsreihenfolge erfordern. Nur wenn all diese Bedingungen erfüllt sind, eignen sich Aufträge für den Freeblock-Scheduler. Diese Einschränkungen werden gemacht, da die Freeblock-Aufträge quasi nebenbei erledigt werden sollen und vorab nicht garantiert werden kann, wann sie zwischen die Hauptaufträge des eigentlichen Schedulers passen, ohne diesen zu behindern.

Diese nötigen Einschränkungen an die Freeblock-Aufträge sind charakteristisch für viele festplattenintensive Hintergrundaufgaben, die üblicherweise nur in Leerlaufzeiten durchgeführt werden. Beispiel dafür sind unter anderem die Suche nach Viren auf der Festplatte oder das Erstellen von Sicherungskopien. Aber auch das Defragmentieren oder Replizieren von Daten um zukünftige Zugriffe zu beschleunigen, sind typische Kandidaten für einen Freeblock-Scheduler. Auch im Zusammenhang mit Hochleistungssystemen, die mit einem Festplattenverbund (RAID) ausgestattet sind, gibt es Aufgaben, für die sich das Freeblock-Scheduling eignet.

Implementierung. Eines der hauptsächlichen Probleme, die es für das Freeblock-Scheduling zu lösen galt, war die sehr exakte Vorhersage der einzelnen Verzögerungskomponenten. Ohne eine präzise Vorhersage dieser Zeiten kann es passieren, dass die Freeblock-Aufträge die Hauptaufträge beeinflussen. Innerhalb der Festplatten sind diese Verzögerungen sehr genau bekannt, aber außerhalb müssen die Zeiten anhand von Beobachtungen bestimmt werden. Um zumindest die physikalische Struktur der Festplatte zu kennen, wurden die gleichen Hilfsmittel eingesetzt wie bei den Track-Aligned Extents aus Abschnitt 3.2 zur Erkennung der Abbildung von LBNs auf Sektoren.

Beim Bestimmen der Verzögerungszeiten gibt es viele Schwierigkeiten, die zu Ungenauigkeiten führen können. So sieht der Scheduler für jeden Auftrag nur die gesamte Bearbeitungsdauer durch die Festplatte, ohne jedoch zu wissen, wie sich diese Bearbeitungszeit zusammensetzt. Darüber hinaus gibt es auch noch mehrere Möglichkeiten, wie die Zeiten zusammengesetzt sein können. So kann es nach der Positionierung des Schreib-/Lesekopfes zum Beispiel passieren, dass eine festplatteninterne Überprüfung der genauen Ausrichtung des Kopfes ergibt, dass die Abweichung von der Sollposition die Toleranzgrenze überschreitet. Dann muss die Kopfposition korrigiert werden muss, was zusätzliche und vor allem unerwartete Zeit kostet. Darüber hinaus gibt es noch weitere festplatteninterne Aktivitäten, die durch die Firmware der Festplatte angestoßen werden, wie zum Beispiel die komplette Rekalibrierung der Festplattenmechanik, die durch Temperatureffekte nötig werden kann. Auch durch Cacheeffekte und andere Optimierungen innerhalb der Festplatte können unerwartete Verzögerungszeiten entstehen. All diese Unbekannten erschweren genaue Vorhersagen, die für den Freeblock-Scheduler sehr wichtig sind.

Aufgrund dieser Schwierigkeiten, die Verzögerungen außerhalb der Festplatte präzise bestimmen zu können, wird üblicherweise eine eher konservative Abschätzung vorgenommen, da durch eine zu optimistische Schätzung der Freeblock-Auftrag im schlimmsten Fall nicht rechtzeitig fertig wird, um den nachfolgenden Hauptauftrag im unmittelbaren Anschluss beginnen zu können. In einem solchen Fall müsste zusätzlich eine komplette Umdrehung abgewartet werden, bis dann mit der Bearbeitung des Hauptauftrags begonnen werden kann.

Ein weiteres Problem, das ein nur schwer vorhersagbares Verhalten der Festplatte bedingt, entsteht durch die aggressiven Optimierungen, die festplattenintern verfolgt werden. Wenn der Freeblock-Scheduler einen Auftrag auswählt, der auf der gleichen Spur lesend zugreift wie der zuvor beendete Hauptauftrag (Abbildung 4 b), so kann es passieren, dass die Festplattenlogik einen sequentiellen Zugriff zu erkennen glaubt, und darum selbstständig weitere Leseoperationen ausführt, um die erwarteten folgenden Leseanfragen aus dem Cache beantworten zu können. Durch solche unnötigen Zugriffe, die eigentlich der Optimierung dienen, geht nicht nur Zeit verloren, sondern der Schreib-/Lesekopf befindet sich im Anschluss daran in einer völlig anderen Position, als von dem Freeblock-Scheduler erwartet. Die anschließende Vorhersage wird darum in den meisten Fällen nicht mit der Realität übereinstimmen. Nach der Beendigung des nächsten Hauptauftrages befindet sich der Kopf aber wieder in einer definierten, dem Freeblock-Scheduler bekannten Position, so dass solche Eigeninitiativen nur lokale Auswirkungen haben.

Diese falsch erkannten sequentiellen Vorabzugriffe können in einigen Fällen vermieden werden. Wenn der Freeblock-Auftrag auf der gleichen Spur beispielsweise mit dem vorangehenden Hauptauftrag zusammengefasst würde, so würde die festplatteninterne Logik nur einen einzigen Zugriff sehen und darum nicht auf ein sequentielles Zugriffmuster schließen. Allerdings wäre dadurch der vorangehende Hauptauftrag erst dann beendet, wenn auch der anschließende Freeblock-Auftrag beendet wäre, wodurch der Hauptauftrag eine Verzögerung erfahren würde. Darum werden Freeblock-Zugriffe auf derselben Spur wie der vorangegangene Auftrag nach Möglichkeit vermieden. Kein Problem ist es hingegen, einen Freeblock-Auftrag, der auf derselben Spur liegt wie der nachfolgende Hauptauftrag, mit diesem zu einer gemeinsamen Anfrage zusammenzufassen.

Der Aufbau eines Freeblock-Schedulers besteht hauptsächlich aus drei Teilen. Ein Teil ist der Scheduler für die Hauptaufträge, der aus einem bestehenden System übernommen werden kann. Dazu kommt der Scheduler, der die Freeblock-Aufträge verwaltet, und schließlich noch eine gemeinsame Warteschlange, in der die Aufträge der beiden Scheduler eingetragen werden, um sie von dort an die Festplatte weiterzugegeben. Der Hauptscheduler sollte dabei immer zwei Aufträge in dieser Warteschlange vorrätig halten: zum einen den Auftrag, der gerade bearbeitet wird, und zum anderen den Folgeauftrag. Durch diese frühzeitige Festlegung auf einen nachfolgenden Auftrag durch den Hauptscheduler hat der Freeblock-Scheduler genügend Zeit, einen passenden Freeblock-Auftrag zwischen den beiden Hauptaufträgen auszuwählen. Nach Möglichkeit wird der ausgewählte Freeblock-Auftrag noch vor der Beendigung des ersten Hauptauftrages an die Festplatte weitergegeben. Dadurch kann die Festplatte sich bereits um den Freeblock-Auftrag kümmern, während der erste Hauptauftrag noch über einen Buszugriff in den Hauptspeicher übertragen wird. Außerdem braucht die Verzögerungszeit, die durch die Busübertragung stattfindet, dann nicht in die Zeitschätzung des Freeblock-Schedulers einbezogen werden. Andererseits sollten sich auch niemals mehr als zwei Aufträge in der festplatteninternen Warteschlange befinden, da sie dann durch die Festplattenlogik umsortiert werden könnten, wodurch im Allgemeinen die Vorhersage des Freeblock-Schedulers nicht mehr zutrifft.

4. Ergebnisse

In vorab durchgeführten Simulationen stellte sich heraus, dass ein Freeblock-Scheduler mit einer perfekten Verzögerungsvorhersage, wie sie nur im Inneren der Festplatte zu erzielen wäre, zwischen 20 und 50 Prozent der theoretisch erreichbaren Bandbreite einer Festplatte, die während des Betriebs dauernd beschäftigt ist, für Hintergrundanwendungen verfügbar machen könnte. Für den implementierten Freeblock-Scheduler, der auf die ungenauere Verzögerungsvorhersage außerhalb der

Festplatte zurückgreifen muss, wird in den Testläufen eine Bandbreite von 3,1 MB/s erreicht, was in etwa 15 Prozent der theoretischen Bandbreite der im Experiment verwendeten Hardware entspricht. Dabei werden die Antwortzeiten für die Festplattenaufträge der Vordergrundanwendungen um weniger als 2 Prozent verlangsamt. Das Hintergrundanwendungsprogramm hat in diesem Experiment nur die Aufgabe, die gesamte Festplatte einzulesen, was für die benutzte 9-GB-Festplatte trotz der hohen Auslastung durch die Vordergrundanwendungen insgesamt 37-mal pro Tag gelingt.

Die erreichte Bandbreite für die Hintergrundanwendungen von 15 Prozent, die im Vergleich zu den erwarteten 20 bis 50 Prozent sehr gering ausfällt, lässt sich zum einen durch die Ungenauigkeit der geschätzten Verzögerungszeiten und der daraus resultierenden konservativen Benutzung des Verfahrens erklären. Zum anderen geht durch den Verzicht auf Freeblock-Zugriffe, die auf derselben Spur wie der vorangegangene Vordergrundzugriff liegen, viel Potential verloren, da gerade diese Zugriffe ohne zusätzliche Kopfpositionierung auskommen würden und somit mehr Sektoren einlesen könnten.

Die zusätzliche Prozessorauslastung von 9 Prozent, die bei den Versuchen gemessen wurde, hält sich gegenüber den 15 Prozent zusätzlicher Bandbreite in Grenzen und es wird auch erwartet, dass die Prozessorauslastung durch Verfeinerungen an den verwendeten Algorithmen deutlich verringert werden kann.

Vor allem für Rechnersysteme, die eine hohe Verfügbarkeit haben müssen und während ihrer gesamten Laufzeit stark beansprucht werden, bietet sich diese Methode an, um die dringend nötigen Wartungsarbeiten neben der eigentlichen Arbeit erledigen zu können.

Literaturverzeichnis

[ID01] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *In Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP), 2001*, 2001.

[LSG02] Christopher R. Lumb, Jiri Schindler, and Gregory R. Ganger. Freeblock scheduling outside of disk firmware. In *In Proc of Conf. on File and Storage Technologies, 2002*, 2002.

[Pat03] David A. Patterson. Storage devices and raid. In *Computer Science 252; Fall 2000*, 2003.

[SGLG02] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-aligned extents: Matching access patterns to disk drive characteristics. In *In Proc. of Conf. on File and Storage Technologies, 2002*, 2002.

[Sta98] William Stallings. *Operating Systems: Internals and Design Principles*. 1998.

KAPITEL 2

Gang Scheduling

Seminarbeitrag von **Philipp Mergenthaler**

Beim gleichzeitigen Ablauf mehrerer paralleler Programme auf Multiprozessoren stellt sich die Frage, ob und wie dies bei der Programmablaufplanung berücksichtigt werden sollte. Es werden verschiedene Verfahren vorgestellt, kommunizierende Prozesse gruppenweise zusammenzufassen und als Prozessgruppe gleichzeitig zum Ablauf zu bringen („Gang-Scheduling“). Sowohl theoretische Abschätzungen und Simulationen als auch praktische Ergebnisse zeigen, dass Gang-Scheduling deutlich bessere Auslastung von Parallelrechnern ergeben kann.

1. Einleitung

Parallele Programme können als Menge gleichzeitig ablaufender Prozesse bzw. Fäden implementiert werden. Kommunikation zwischen solchen Prozessen findet man z.B. bei Programmen, die große Berechnungen auf mehrere Prozessoren verteilen. Durch die Verteilung wird im Allgemeinen Kommunikation zwischen den Prozessen des parallelen Programms notwendig. Sie bilden daher eine zusammengehörige Gruppe interagierender Prozesse. Häufig wird eine solche Gruppe von zusammengehörenden, miteinander kommunizierenden Prozessen nicht allein auf einem Rechner laufen, sondern zusammen mit anderen Prozessen oder Prozessgruppen. Programmablaufplanung, die nicht berücksichtigt, welche Prozesse miteinander kommunizieren und wie fein die Granularität der Kommunikation ist, kann die Performanz beeinträchtigen. Im Folgenden werden im Wesentlichen Systeme betrachtet, bei denen die Kommunikation nachrichtenbasiert (message passing) erfolgt.

Bei der Kommunikation zwischen zwei Prozessen werden in der Regel die Sende- und Empfangsoperation zu verschiedenen Zeitpunkten angestoßen werden. Dies kann dazu führen, dass Prozesse warten müssen. Ein Beispiel ist das Senden von Daten von einem Prozess zu einem anderen mit synchroner Empfangsoperation. Falls die Empfangsoperation vor der Sendeoperation angestoßen wird, muss der empfangende Prozess warten, bis die Daten gesendet wurden. Ein anderes Beispiel ist ein Rendezvous zwischen zwei Prozessen; hier muss der zuerst ankommende Prozess warten, bis der zweite die Rendezvous-Stelle erreicht.

Ein durch eine Kommunikationsoperation aufgehaltener Prozess kann also erst weiter ablaufen, nachdem sein Partner zum Ablauf gebracht wurde und seinen Anteil an der Kommunikationsoperation erledigt hat. Durch das ständige **Blockieren**

kommt es zu einer großen Anzahl von Prozesskontextwechseln, die viel Zeit kosten können. Alternativ zum Blockieren bietet sich das **aktive Warten** oder **busy waiting** an. Dabei belegt der wartende Prozess seinen Prozessor weiterhin. Aktives Warten ist auf Einprozessorsystemen nicht sinnvoll, weil während des Wartens kein anderer Prozess vorankommen kann. Auf Mehrprozessorsystemen ist es eventuell sinnvoll, falls die Wartezeit kürzer ist als die Prozesswechselzeiten, die bei einem Kontextwechsel zusätzlich anfallen würden.

Auf einem Parallelrechner wird man typischerweise mehrere Programme gleichzeitig laufen lassen wollen, um die Rechenleistung möglichst gut zu nutzen. Damit stellt sich die Frage, wie die Rechenleistung aufgeteilt werden soll. Die zwei grundlegenden Möglichkeiten sind die räumliche und die zeitliche Partitionierung.

Bei der **räumlichen Aufteilung** eines Parallelrechners (**space sharing**) wird der Rechner in kleinere Teilsysteme aufgeteilt, auf denen unabhängig voneinander Programme zum Ablauf gebracht werden können. Dieses Verfahren hat den Vorteil, die Zahl der Kontextwechsel zu minimieren. Allerdings kann es zu schlechten Antwortzeiten führen.

Bei der **zeitlichen Aufteilung** (**time sharing**) wird die Rechenleistung über die Zeit aufgeteilt. Dadurch kann man gute Antwortzeiten erreichen. Ein Beispiel für zeitliche Aufteilung ist das Zeitscheibenverfahren.

Es stellt sich die Frage, ob man durch Kombinieren beider Verfahren die Nachteile vermeiden oder zumindest abschwächen kann.

Im folgenden Abschnitt wird zunächst Gang-Scheduling als Kombination von räumlicher und zeitlicher Partitionierung eingeführt. In Abschnitt 3 wird untersucht werden, wann hierbei aktives Warten besser als blockierendes Warten ist. Danach wird ein modifiziertes Gang-Scheduling-Verfahren vorgestellt und zum Schluss der Einsatz von Gang-Scheduling in der Praxis betrachtet.

2. Gang Scheduling

Falls die Ablaufplanung dazu führt, dass nur ein Teil der Prozesse einer Gruppe von kommunizierenden Prozessen gleichzeitig ablaufen, kann es passieren, dass ein oder mehrere der ablaufenden Prozesse mit den übrigen Prozessen derselben Gruppe kommunizieren wollen und deshalb warten müssen. Wenn dann, nach Ablauf der Zeitscheibe, die nicht ablaufenden und die aktiven Prozesse tauschen, können die Kommunikationsoperationen zwar abgeschlossen werden, aber die jetzt ablaufenden Prozesse werden wahrscheinlich bei weiteren Kommunikationsoperationen auf die jetzt inaktiven Prozessen warten müssen. Dies führt zu zusätzlichen Kontextwechseln und Wartezeiten. Diese Situation, in der kommunizierende Prozesse jedes Mal dann von ihren Prozessoren verdrängt werden, wenn andere Prozesse aus dieser Gruppe zum Ablauf gebracht werden, bezeichnet man als **Prozessflattern**.

Die Idee beim **Gang Scheduling** ist nun, zusammengehörende (intensiv miteinander kommunizierende) Prozesse gleichzeitig auszuführen[**Ous82**]. Dazu ist es zunächst nötig, solche Gruppen zusammengehörender Prozesse zu bestimmen. Sodann muss bei der Programmablaufplanung dafür gesorgt werden, dass solche Prozesse gleichzeitig ausgeführt werden.

Man kann Gang Scheduling als Kombination von räumlicher und zeitlicher Partitionierung betrachten: die Prozesse bzw. Threads einer Gruppe werden mittels space sharing auf dem Rechner verteilt und gleichzeitig ausgeführt, und die Gruppen teilen sich mittels time sharing den Rechner.

2.1. Annahmen und Ziel. Für den Rest von Abschnitt 2 gelten folgende Definitionen und Annahmen:
Mehrere Prozesse, die zusammen ausgeführt werden sollen, werden als **Prozessgruppe** bezeichnet. Prozessgruppen sind statisch und vom Programmierer

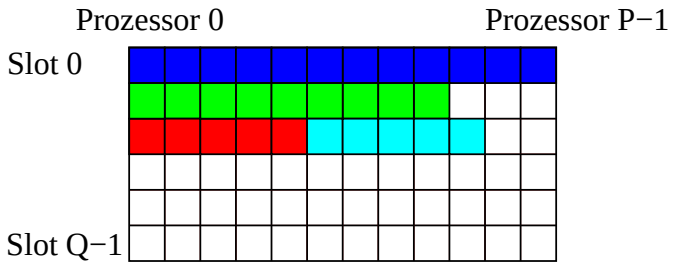


ABBILDUNG 1. Allokation beim Matrix-Algorithmus

festgelegt. Jeder Prozess gehört zu genau einer Prozessgruppe.

Eine Prozessgruppe (bzw. ihre Prozesse) werden als **koordiniert** (coscheduled) bezeichnet, wenn alle ausführbaren Prozesse der Gruppe gleichzeitig auf verschiedenen Prozessoren ablaufen. Falls nicht alle ausführbaren Prozesse einer Gruppe gleichzeitig ablaufen, wird die Gruppe als **fragmentiert** bezeichnet. Als Ziel der Programmablaufplanung wird angenommen, dass die durchschnittliche Zahl von Prozessoren, auf denen koordinierte Prozesse ablaufen, maximal sein soll. Ein Prozess kann jedem beliebigen Prozessor zugeordnet werden und diese Zuordnung kann nicht geändert werden.
Das betrachtete System hat P Prozessoren, die zwischen je Q Prozessen wechseln können. Damit gibt es $P * Q$ **Kacheln**, die wir als **Prozessraum** bezeichnen wollen. Eine Prozessgruppe kann maximal P Prozesse haben. Beim **Allokieren** einer Prozessgruppe wird jeder ihrer Prozesse einer freien Kachel zugewiesen. Q sei so groß, dass Allokierung immer möglich ist.

2.2. Koordinierungs-Algorithmen. Im Folgenden werden drei Algorithmen betrachtet, die den Prozessraum auf unterschiedliche Weisen aufteilen. Für jeden Algorithmus wird zunächst die Allokierung betrachtet (d.h. welche Prozessoren einer Prozessgruppe zugeteilt werden), danach das Scheduling (d.h. welche Prozessgruppen zum Ablauf gebracht werden sollen).

Der Matrix-Algorithmus. Der Matrix-Algorithmus betrachtet den Prozessraum als $Q \times P$-Matrix.

Allokation. Suche die erste Zeile, die genug freie Kacheln hat, dass die gesamte Prozessgruppe hineinpasst.

Scheduling. Im ersten Zeitschritt werden die Prozesse der ersten Zeile zum Ablauf gebracht, im nächsten Zeitschritt die der nächsten Zeile, etc. (round-robin). Falls in der aktuellen Zeile eine oder mehrere Kacheln frei sind oder Prozesse enthalten sind, die nicht ausführbar sind, sucht jeder der betroffenen Prozessoren den nächsten ausführbaren Prozess in seiner Spalte der Matrix und führt ihn aus (sog. Alternative).

Die Vorteile des Matrix-Algorithmus sind seine Einfachheit und die Trennung von globalem und lokalem Scheduling: global wird nur die aktuelle Zeile festgelegt, die Auswahl von Alternativen ist lokal.

Allerdings hat dieser Algorithmus deutliche Nachteile. Zum einen können Prozesse einer Gruppe nicht über mehrere Zeilen verteilt werden, wodurch es oft mehrere freie Kacheln pro Zeile gibt (interner Verschnitt). Zum anderen erfolgt die Auswahl von Alternativen auf jedem Prozessor unabhängig von den anderen. Dadurch werden die Alternativen meist als Fragmente laufen, weshalb sie ihrerseits bei einer Kommunikationsoperation blockieren werden. Die folgenden beiden Algorithmen sollen das erste Problem (interner Verschnitt) vermeiden.

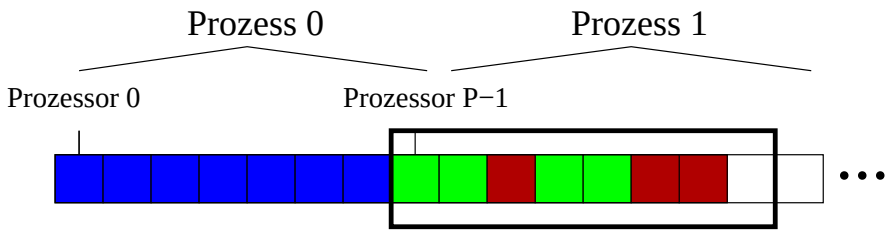


ABBILDUNG 2. Allokation beim kontinuierlichen Algorithmus

Der kontinuierliche Algorithmus. Der kontinuierliche Algorithmus betrachtet den Prozessraum als eine Reihe von Kacheln, in der die Zeilen der Matrix aneinander gehängt werden. Ein „Fenster“ von P benachbarten Kacheln enthält damit genau eine Kachel von jedem Prozessor.

Allokation. Betrachte ein Fenster von P benachbarten Kacheln am Anfang des Prozessraums. Falls es nicht genug freie Kacheln für die aktuelle Arbeitsgruppe gibt, verschiebe das Fenster so weit nach rechts, dass die erste Kachel des Fensters frei ist und sich links von ihr eine belegte Kachel befindet. Wiederhole dies, bis das Fenster einen Abschnitt im Prozessraum erreicht, der die gesamte Arbeitsgruppe aufnehmen kann.

Scheduling. Zu Beginn eines Durchgangs platziere ein Fenster von P benachbarten Kacheln am Anfang des Prozessraums. Verschiebe das Fenster nach rechts, bis seine erste Kachel den ersten Prozess einer Gruppe enthält, die im aktuellen Durchgang noch nicht koordiniert zum Ablauf gebracht wurde, und bringe sie zum Ablauf. Wiederhole dies, bis alle Prozesse mindestens einmal zum Ablauf gebracht wurden und beginne dann den nächsten Durchgang. Für leere Kacheln oder Kacheln mit blockierten Prozessen werden, ähnlich wie beim Matrix-Algorithmus, alternative Prozesse ausgewählt.

Mit der Bedingung, das Fenster zu einer Gruppe zu verschieben, die noch nicht koordiniert zum Ablauf gebracht wurde, erreicht man, dass alle Gruppen möglichst gleich oft koordiniert ablaufen. Anderenfalls würden kleine Arbeitsgruppen begünstigt.

Der kontinuierliche Algorithmus nutzt den Prozessraum effizienter als der Matrix-Algorithmus. Sein Hauptnachteil liegt darin, dass die Prozesse einer Gruppe nicht immer in benachbarten Kacheln liegen. Wenn im Prozessraum kleine „Löcher“ von freien Kacheln entstehen, kann es passieren, dass eine Arbeitsgruppe mit wenigen Prozessen auf mehrere Löcher verteilt wird und dadurch seltener koordiniert abläuft als eine unzerteilte Arbeitsgruppe. Der letzte hier vorgestellte Algorithmus ist ein Versuch, diese Problem zu vermeiden.

Der unzerteilte Algorithmus. Der unzerteilte Algorithmus unterscheidet sich vom kontinuierlichen Algorithmus nur darin, dass Gruppen unzerteilt allokiert werden, d. h. in einer durchgehenden Reihe von Kacheln.

Dieser Algorithmus nutzt zwar den Prozessraum weniger effizient als der kontinuierliche Algorithmus (d.h. es gibt mehr Löcher, also Verschnitt), erreicht aber durch das Vermeiden von Fragmentation ein deutlich besseres Verhalten unter hoher Last.

2.3. Vergleich durch Simulation. Die drei Algorithmen wurden in Simulationen (mit $P = 50$) verglichen. Die Parameter des Simulationsmodells sind:

- Größe der Arbeitsgruppe
- Last
- Lebensdauer der Gruppen
- Leerlaufanteil (idle fraction)

Gemessen wurde die Effektivität des Coscheduling, d.h. der Durchschnitt des Verhältnisses von Prozessoren auf denen koordinierte Prozesse ablaufen zu allen Prozessoren mit laufenden Prozessen. Abbildung 3 zeigt die Effektivität in Abhängigkeit

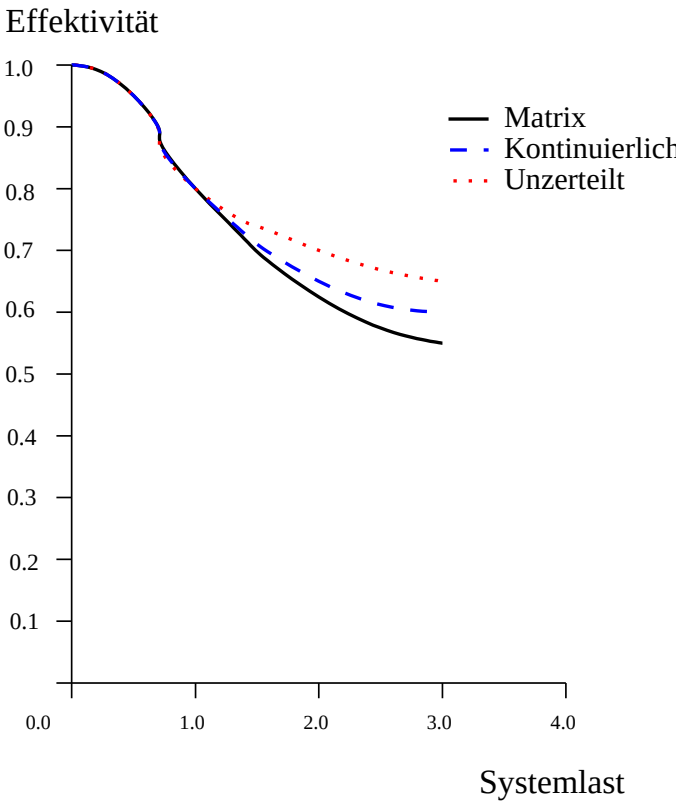


ABBILDUNG 3. Effektivität in Abhängigkeit von der Systemlast

von der Last des Systems. Während bei geringer Last alle drei Algorithmen eine Effektivität nahe 1.0 erreichen, sinkt sie bei höherer Last ab. Gründe dafür sind:

- beim kontinuierlichen und unzerteilten Algorithmus führt eine Arbeitsgruppe mit weniger als P Prozessen dazu, dass ein Teil der nächsten Gruppe als Fragment ausgeführt wird,
- das Auswählen von Alternativen geschieht unkoordiniert, so dass diese Prozesse oft als Fragment ablaufen.

Die Unterschiede in der Leistung der drei Algorithmen sind relativ gering, was sich damit erklären lässt, dass die Auswahl von Alternativen bei allen gleich funktioniert.

Abbildung 4 zeigt die Coscheduling-Effektivität in Abhängigkeit von der Größe der Arbeitsgruppen. Der kontinuierliche Algorithmus verhält sich bei kleinen Gruppengrößen am schlechtesten, da hier die Arbeitsgruppen i. a. über viele kleine Löcher verteilt werden. Der Matrix-Algorithmus zeigt seine Schwäche bei Gruppengrößen von ca. $P/2$, da er hier die Prozesse nicht dicht in den Prozessraum packen kann.

3. Evaluierung von Gang-Scheduling bei feinkörniger Kommunikation

3.1. Modellierung. In diesem Abschnitt wird das Verhalten von Gang-Scheduling bei feinkörniger Kommunikation untersucht [**FR92**]. Insbesondere soll untersucht werden, wann Blockieren bzw. aktives Warten besser ist.

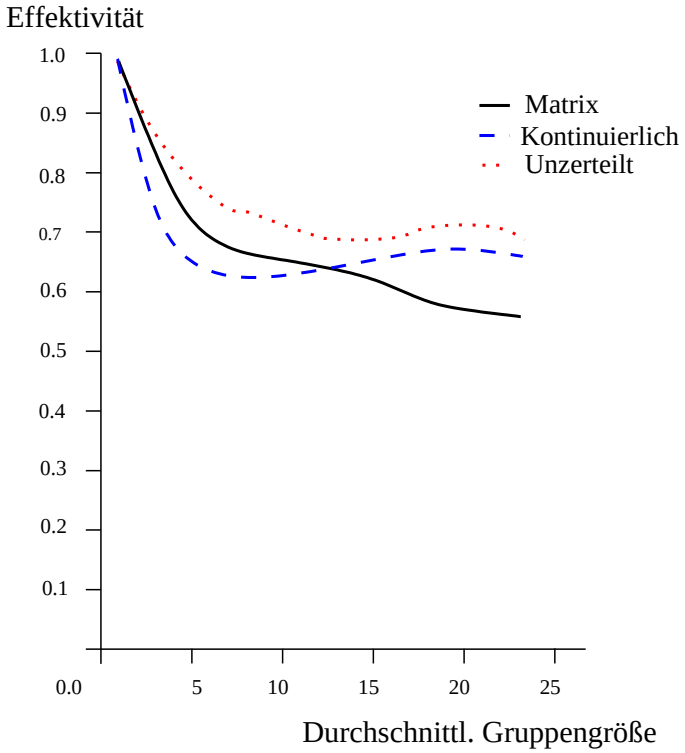


ABBILDUNG 4. Effektivität in Abhängigkeit von der Gruppengröße

Für die folgenden Betrachtungen bestehe ein Programm aus n kommunizierenden Prozesse, wobei n kleiner oder gleich der Anzahl der Prozessoren sei und jeder Prozess auf einem eigenen Prozessor laufe. Die Prozesse seien iterativ. In jeder Iteration rechne jeder Prozess für eine gewisse Zeit (ohne Kommunikation), dann folge eine Barrieresynchronisation aller Prozesse.

Dieses Modell repräsentiert viele parallele Algorithmen, z.B. Simulationen, die auf der numerischen Lösung partieller Differentialgleichungen beruhen und zelluläre Automaten.

Im Folgenden werden nicht die Rechen- und Wartezeit jedes einzelnen Prozesses betrachtet, sondern nur die durchschnittlichen Zeiten. Die durchschnittliche Rechenzeit pro Iteration sei t_p, die durchschnittliche Wartezeit t_w. Die Zeit für die Barrieresynchronisation – die bei allen Prozessen auftritt – wird in t_p einbezogen. Im Fall dass alle Prozesse die Barriere gleichzeitig erreichen, ist t_w damit Null.

Die Zahl der Slots pro Prozessor sei l, wobei hier nur eine Prozessgruppe betrachtet wird. Der Scheduler sei fair, d.h. alle Gruppen werden gleich behandelt. Die Dauer einer Zeitscheibe sei τ_q. Die Dauer eines Kontextwechsels sei τ_{cs}, mit $\tau_{cs} \ll \tau_q$. Blockierendes Warten dauere $\alpha * \tau_{cs}$ wobei α konstant und etwas größer als 1 sei.

Zunächst werde aktives Warten mit Gang-Scheduling betrachtet. Für k Iterationen ergibt sich die Gesamtzeit

$$T = \left(k(t_p + t_w) + \frac{k(t_p + t_w)}{\tau_q} \tau_{cs} \right) l. \tag{1}$$

Sie setzt sich zusammen aus der Zeit fürs Rechnen und Warten auf die Synchronisation ($k(t_p + t_w)$) und der Zeit für die Kontextwechsel, von denen pro Zeitscheibe τ_q einer stattfindet.

Die durchschnittliche Zeit pro Iteration ist damit

$$t = \left(1 + \frac{\tau_{cs}}{\tau_q}\right)(t_p + t_w)l. \tag{2}$$

Wegen des Faktors $t_p + t_w$ hängt die Dauer vom langsamsten Prozess ab. Bei feinkörniger Kommunikation ($t_p + t_w < \tau_q$) verteilen sich die Kosten für einen Kontextwechsel auf mehrere Iterationen.

Nun werde blockierendes Warten mit unkoordinierter Programmablaufplanung betrachtet. Ein Prozess, der auf seinen Kommunikationspartner warten muss, wird in diesem Fall vom Prozessor verdrängt.

Hier sind zwei Fälle zu unterscheiden: grob- und feinkörnige Kommunikation. Bei grobkörniger Kommunikation findet Blockieren nur selten statt, Kontextwechsel fallen fast nur bei abgelaufener Zeitscheibe an. Unter der Annahme, dass alle $l * n$ Prozesse iterativ sind, ergibt sich als Gesamtzeit

$$T = \left(k(t_p + t_w) + \frac{k(t_p + t_w)}{\tau_q}\tau_{cs}\right)\frac{t_p}{t_p + t_w}l \tag{3}$$

und die Zeit pro Iteration ist

$$t = \left(1 + \frac{\tau_{cs}}{\tau_q}\right)t_p l. \tag{4}$$

Im Vergleich zu Gleichung (2) fällt die Wartezeit weg.

Bei feinkörniger Kommunikation treten Kontextwechsel hauptsächlich aufgrund von blockierendem Warten auf, was mit dem Aufwandsfaktor α berücksichtigt wird. Die Gesamtzeit ist

$$T = k\left(t_p + \frac{n-1}{n}\alpha\tau_{cs}\right)l. \tag{5}$$

Damit ist die durchschnittliche Zeit für eine Iteration

$$t = k\left(1 + \frac{(n-1)\alpha\tau_{cs}}{nt_p}\right)t_p l. \tag{6}$$

Auch hier beträgt die effektive Dauer einer Iteration nur t_p. Allerdings steht im Nenner des Vorfaktors jetzt t_p, und für feinkörnige Kommunikation gilt $t_p << t_q$.

3.2. Experiment. Um das Modell zu bestätigen wurden einige Experimente auf dem Makbilan-Multiprozessor durchgeführt. Dies ist ein NUMA-Computer mit 10 Prozessoren. Abbildung 5 zeigt, dass die Messergebnisse die Erwartungen bestätigen. Bei grobkörniger Kommunikation erfordert blockierendes Warten weniger Zeit pro Iteration, während bei feinkörniger Kommunikation Gang-Scheduling mit aktivem Warten zu schnellerem Ablauf des Programms führt.

3.3. Eignung in verschiedenen Situationen. Um genauer abzuschätzen, in welcher Situation sich welches Vorgehen eignet, werden t_{AW}, die Zeit pro Iteration bei Gang-Scheduling mit aktivem Warten, und t_{BLK}, die Zeit bei blockierendem Warten betrachtet. Diese Werte werden durch Gleichung 2 bzw. Gleichung 6 gegeben.

Der schattierte Bereich links unten in Abbildung 6 stellt den Bereich dar, in dem aktives Warten deutlich besser (mindestens Faktor $\sigma = 2$) als blockierendes Warten ist, während der schattierte Bereich links oben den Bereich angibt, in dem blockierendes Warten besser ist. Wenn man den Faktor σ auf 1 reduziert, konvergieren die beiden Bereiche und werden dann durch eine Gerade mit sehr kleiner negativer Steigung getrennt.

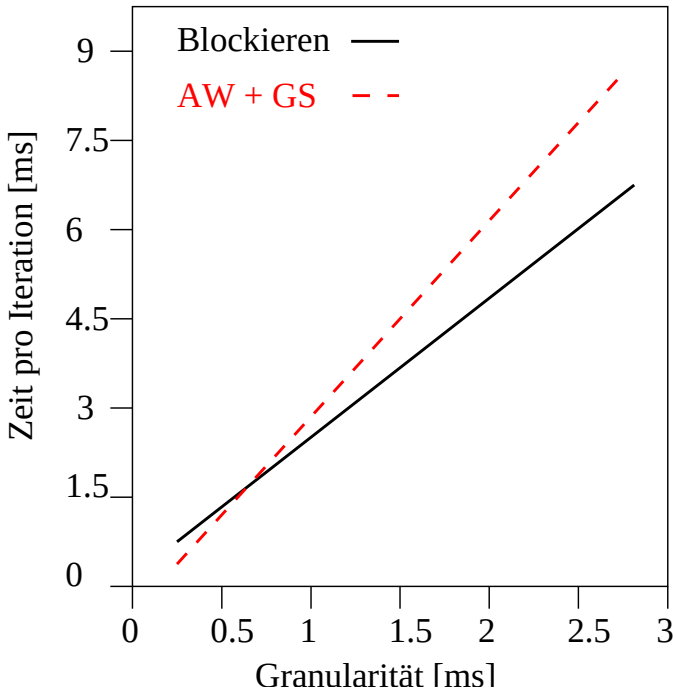


ABBILDUNG 5. Gesamtzeit pro Iteration bei blockierendem und aktivem Warten

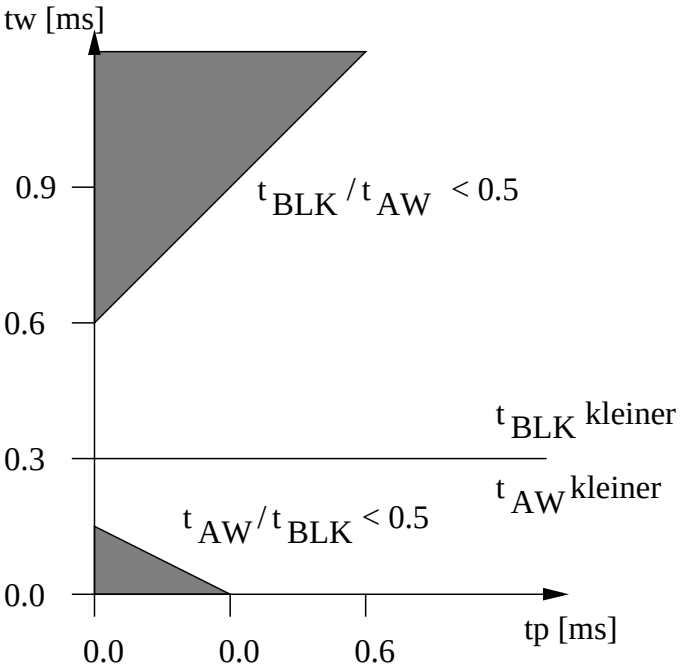


ABBILDUNG 6. Vergleich der relativen Leistung bei aktivem und blockierendem Warten im Phasendiagramm

Die Werte in diesem Diagramm beziehen sich auf den Makbilan-Computer mit $\tau_q = 50$, $\tau_{cs} = 0.2$ und $\alpha = 1.5$.

Generell kann festgehalten werden, dass aktives Warten bei feinkörniger Kommunikation (t_p klein) und ausgewogener Verteilung der Arbeit auf die Prozesse (t_w

klein) von Vorteil ist, während blockierendes Warten bei stark unausgewogener Arbeitsverteilung von Vorteil ist.

4. Ein verbesserter Algorithmus

Dieser Abschnitt stellt einen verbesserten Algorithmus vor, der zu höherer Auslastung und reduzierter Fragmentierung führt, dazu aber eine bestimmte Eigenschaft des Programms voraussetzt: Formbarkeit. Formbarkeit ist die Eigenschaft eines Programms, den Grad seiner Parallelität zur Laufzeit zu ändern und an die Zahl der ihm zugewiesenen Prozessoren anzupassen. Ein Beispiel für ein Programmiermodell mit dieser Eigenschaft ist OpenMP, das dem fork-join-Modell folgt. Ein solches Programm startet als ein einzelner Faden bis es auf ein paralleles Programmkonstrukt stößt. Dann wird ein Team von Fäden erzeugt, die jeder für sich den Programmabschnitt innerhalb des Programmkonstrukts abarbeiten und an deren Ende sich mit dem Hauptfaden synchronisieren und beendet werden. Die Zahl der Fäden ist nicht vom Programm festgelegt und kann pro Programmkonstrukt anders sein.

4.1. Performance-Driven Gang Scheduling. In der Praxis beobachtet man, dass Programme oft vom Benutzer mit der Anzahl von Prozessoren gestartet werden, die den besten **Speedup** (Beschleunigung, die Zeit für die Ausführung auf einem Prozessor geteilt durch die Zeit für die Ausführung auf mehrere Prozessoren) ergibt. Allerdings nimmt dabei typischerweise die **Effizienz** (Speedup geteilt durch Anzahl der Prozessoren) ab (d.h. bei Benutzung von n Prozessoren ist der Speedup kleiner als n). Dies ist aus Sicht des Benutzers kein Problem, aber um eine möglichst hohe Auslastung des Rechners insgesamt zu erreichen, sollten Programme nur mit einer Anzahl von Prozessoren laufen, die sie noch effizient nutzen können. Man hat in dieser Situation also konträre Interessen zwischen den einzelnen Benutzern und dem Betriebssystem. Um eine möglichst hohe Auslastung des Rechners zu erreichen, sollte die Programmablaufplanung nun also nicht nur den Wunsch des Benutzers sondern auch Laufzeitmessungen des Programms berücksichtigen.

Dies soll mit **Performance-Driven Gang Scheduling** (PDGS) [**CML01**] erreicht werden. Bei diesem Verfahren wird die Leistung beim Abarbeiten eines parallelen Konstrukts zur Laufzeit bestimmt. Es werden hier nur **iterative parallele Anwendungen** betrachtet. Diese haben eine äußere sequentielle Schleife, die parallele Konstrukte (z.B. Schleifen) enthält.

Wenn der Programmablauf eine parallele Schleife erreicht, werden zunächst einige Durchläufe dieser Schleife auf nur einem Prozessor abgearbeitet, um einen Referenzwert zu erhalten. Der Rest der Schleife wird auf allen verfügbaren Prozessoren abgearbeitet und der Speedup bestimmt [**CL99**].

Die dazu nötigen Aufrufe der Messroutinen können automatisch vom Übersetzer anhand der OpenMP-Direktiven eingesetzt werden:

- init_parallelism, wird vor der äußeren Schleife aufgerufen und initialisiert die benutzten Datenstrukturen.
- open_parallel_region, wird zu Beginn jedes parallelen Konstrukts aufgerufen. Hier wird die Zeit bestimmt und die Anzahl der Prozessoren (1 oder P) festgelegt.
- close_parallel_region, wird am Ende jedes parallelen Konstrukts aufgerufen. Hier wird wieder die Zeit bestimmt und der Speedup berechnet.
- end_parallelism, wird nach dem Ende der äußeren Schleife aufgerufen.

Es ist notwendig, dass die benutzte Parallelbibliothek es dem Programm erlaubt, die gewünschte Zahl der Prozessoren anzugeben. Darüber hinaus muss die Parallelbibliothek dem Scheduler den Speedup mitteilen.

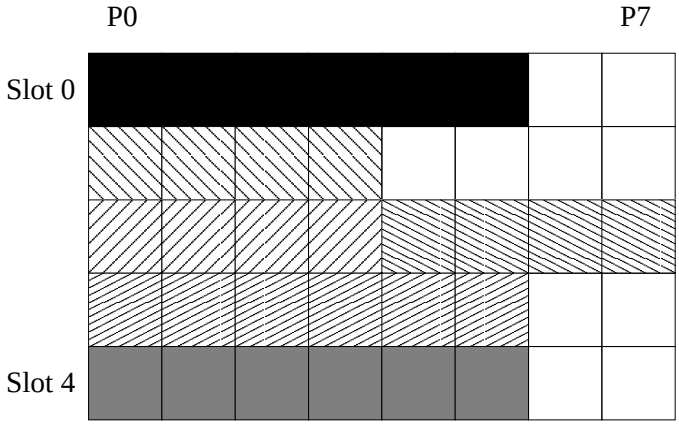


Abbildung 7. Matrix für ein System mit 8 Prozessoren

Der PDGS-Algorithmus hat große Ähnlichkeit mit dem Matrix-Algorithmus (siehe Abbildung 7); die Allokation erfolgt wie beim Matrix-Algorithmus. Das Scheduling ist etwas aufwändiger (vgl. Abbildung 8): Nach Ablauf einer „time sharing“-Zeitscheibe werden die Prozesse der nächsten Zeile zum Ablauf gebracht. Zusätzlich werden innerhalb einer „time sharing“-Zeitscheibe mehrmals die Prozessoren neu auf die aktiven Prozessgruppen verteilt, wobei die Messwerte berücksichtigt werden. Dadurch wird erreicht, dass die Prozessoren effizient genutzt werden. Die „time sharing“-Zeitscheibe ist ein Vielfaches der „space sharing“- Zeitscheibe. Das Ziel dieser Aufteilung ist es, die Zahl der Kontextwechsel zu verringern, aber die Prozessorallokation mit höherer Zeitauflösung als der „time sharing“-Zeitscheibe durchzuführen.

```
while(1){
  time+=space_sharing_quantum;
  if ((time%time_sharing_quantum)==0){
      Change_slot();
      Restore_state();
  }
  Distribute_processors();
  sleep(space_sharing_quantum);
}
```

Abbildung 8. Zentrale Schleife des Schedulers bei PDGS

4.2. Compress&Join-Algorithmus. Durch den Einsatz von PDGS erreicht man eine gute Auslastung des Rechners. Auch verringert sich eventuell die Zahl der belegten Zeilen der Matrix. Es kann aber immer noch Fragmentierung auftreten. Da aber Formbarkeit vorausgesetzt wurde, kann man sie jetzt noch ausnutzen, um Fragmentierung zu vermeiden.

Das soll mit dem **Compress&Join-Algorithmus** erreicht werden [**CML01**]. Auch dieser Algorithmus benutzt eine Matrix analog zum Matrix-Algorithmus. Die Größe der Arbeitsgruppen wird hier aber so angepasst, dass sie kompakt in die Matrix eingefügt werden können, wodurch Verschnitt vollständig vermieden wird. Außerdem wird dadurch die Zahl der Kontextwechsel zum Umschalten auf alternative Prozessoren reduziert. Zum Beispiel zeigt Abbildung 7 den Verschnitt, der beim Matrix-Algorithmus dadurch entsteht, dass die ersten beiden Prozessgruppen

zwar jeweils nicht alle Prozessoren belegen, aber doch so viele, dass sie nicht gemeinsam in eine Zeile passen. Der Compress&Join-Algorithmus dagegen reduziert

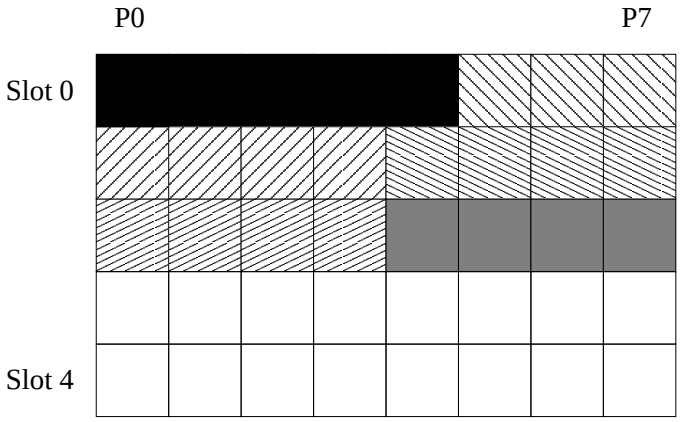


ABBILDUNG 9. Die mittels Compress&Join-Algorithmus belegte Matrix

die Zahl der Prozesse der beiden Prozessgruppen so weit, dass sie gleichzeitig ablaufen können (siehe Abbildung 9). Nach Anwendung des Compress&Join-Algorithmus haben diese Prozessgruppen (und auch die Gruppen aus den letzten beiden Zeilen von Abbildung 7) also weniger Prozessoren als vorher zur Verfügung. Gleichzeitig sinkt aber die Zahl der belegten Zeilen von fünf auf drei, wodurch die Zahl der Kontextwechsel verringert wird. Es wird angenommen, dass der Gewinn durch die geringere Zahl von Kontextwechseln höher ist als der Verlust durch die geringere Zahl von Prozessoren pro Programm.

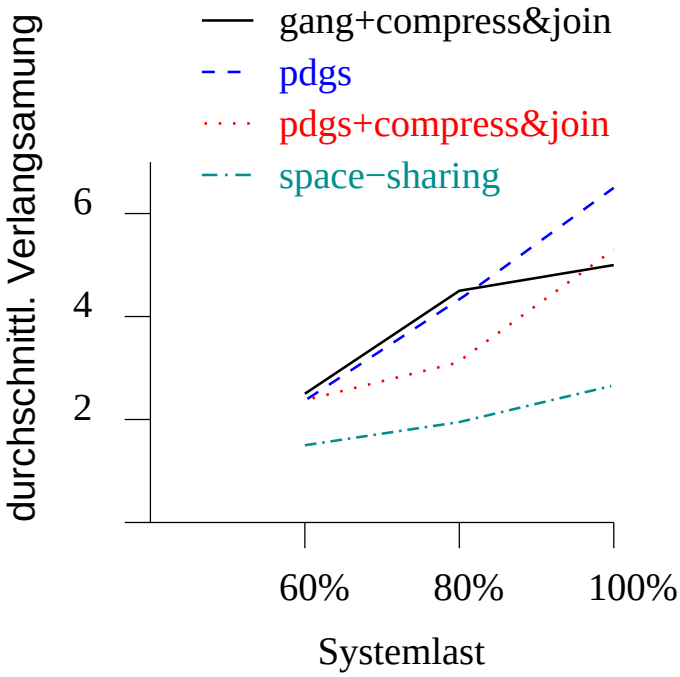


ABBILDUNG 10. Vergleich von GS, PDGS und Compress&Join-Algorithmus

Messergebnisse auf einem 64-Prozessor-System (siehe Abb. 10) zeigen, dass Compress&Join sowohl zusammen mit Gang-Scheduling als auch zusammen mit

PDGS eine Verbesserung bewirkt. Die Verlangsamung ist relativ zu einem Programmlauf allein, mit der Zahl von Prozessoren, die den höchsten Speedup ergeben, angegeben. Zum Vergleich ist ein Space-Sharing-Verfahren aufgeführt, das, ähnlich wie PDGS, zur Laufzeit die von den Anwendungen erreichte Effizienz misst und berücksichtigt. Es ist den Gang-Scheduling-Algorithmen überlegen, da bei ihm keine Kontextwechsel auftreten.

5. Einsatz von Gang-Scheduling in der Praxis

Am Lawrence Livermore National Laboratory (LLNL) wurden zwischen 1989 und 1997 Gang Scheduler für drei verschieden Rechner implementiert [**Jet97**]. Zwischen diesen Implementierungen gibt es zum Teil beträchtliche Unterschiede; diese sind teilweise durch die unterschiedlichen Rechnerarchitekturen bedingt, stellen aber auch Verbesserungen aufgrund der Erfahrungen mit früheren Implementierungen dar.

5.1. Anforderungen und Designstrategie. Die meisten auf diesen Rechnern laufenden Jobs sind groß; z.B. belegen auf der Cray T3D mit 256 Prozessoren gut 80% der Jobs 64 und mehr Prozessoren. Ca. zwei Drittel aller Jobs interagieren mit dem Anwender, verbrauchen aber nur ca. 13% der gesamten CPU-zeit. Die meisten Programme benutzen zur Kommunikation die PVM- oder MPI-Bibliothek.

Die Jobs haben unterschiedliche Anforderungen, z.B. in Hinsicht auf Antwortzeit. Deshalb wurden unterschiedliche Klassen eingeführt, die vom Scheduler gesondert behandelt werden. Standardmäßig stehen die folgenden Klassen zur Auswahl:

- Mit dem Anwender interagierende Jobs erfordern gute Antwortzeit und guten Durchsatz während der Arbeitszeit, während sie außerhalb der Arbeitszeit zurückgestuft werden können.
- Debug-Jobs haben ähnliche Anforderungen; sie können auf der Cray T3D nicht von anderen Jobs verdrängt werden.
- Produktionsjobs erfordern keine kurze Antwortzeit, sollten aber außerhalb der Arbeitszeit guten Durchsatz haben.

Darüber hinaus gibt es noch Klassen für z.B. Express-Jobs und Messungen.

Jede Klasse hat einige Scheduling-Parameter wie z.B. einen Prioritätswert. Daneben gibt es einige systemweit geltende Parameter für den Scheduler wie z.B. die Gesamtzahl der Prozessoren die vom Gang-Scheduler verwendet werden dürfen. Diese Parameter können jederzeit verändert werden. Am LLNL werden sie während der Arbeitszeit so eingestellt, dass kurze Antwortzeiten erreicht werden, und sonst so, dass hoher Durchsatz erreicht wird.

5.2. Fallstudie: BBN TC2000. Der Gang-Scheduler für die TC2000 wurde als Dämon implementiert, der beim Start des Systems alle Ressourcen reserviert. Die obligatorische Bibliothek mit den Routinen zur Initialisierung paralleler Programme wurde so angepasst, dass sie sich Ressourcen nicht direkt vom Betriebssystem sondern vom Gang-Scheduler besorgt. Die Anwendungen selbst müssen nicht geändert werden, um den Gang-Scheduler zu nutzen. Der Scheduler steuert den Ablauf, indem er Signale an jeweils alle Prozesse einer Gruppe schickt. Ein Programm kann die Zahl seiner Prozesse zur Laufzeit ändern, indem es den Gang-Scheduling-Dämon benachrichtigt. Dieser Scheduler erreicht einen guten Durchsatz, hat aber auch Nachteile, z.B. begrenzte Ansprechbarkeit (bedingt durch die Zeitscheibe von 10 s).

5.3. Fallstudie: Cray T3D. Der Gang-Scheduler für die Cray T3D wurde ebenfalls als Dämon implementiert, weist aber einige Unterschiede zu dem für die

BBN TC2000 auf. Statt eine feste Zeitscheibe zu verwenden läuft dieser Scheduler ereignisgesteuert. Des Weiteren berücksichtigt er die spezielle Kommunikationshardware der Cray T3D, die einschränkt, welche Prozessoren zusammen für ein Programm allokiert werden können. Die Zahl der Prozesse eines Programms kann zur Laufzeit nicht geändert werden.
Anwendungen werden auf der Cray T3D mittels eines bestimmten Programms gestartet; dieses wurde um die Kommunikation mit dem Gang-Scheduler ergänzt. Dadurch brauchen auch hier die Anwendungen nicht geändert zu werden.
Mit diesem Scheduler wurde eine CPU-Auslastung von über 96% im Wochendurchschnitt erreicht, während der ursprüngliche Scheduler des Betriebssystems (UNICOS MAX) nur ca. 45% erreichte. Die CPU-Auslastung bezeichnet den Zeitanteil, während dessen die CPU ein Programm abarbeitet. Sie wird beeinträchtigt durch:

- Kontextwechsel,
- Prozessoren, die aufgrund der Kommunikationshardware nicht genutzt werden können,
- zu wenig bereitstehende Arbeit (Jobs).

Neben der Auslastung interessiert auch das Antwortzeitverhalten, das als gut beurteilt wurde. Mit dem Anwender interagierende Jobs, die während der Arbeitszeit gestartet wurden, wurden typischerweise innerhalb von 30 Sekunden zum Ablauf gebracht und nicht verdrängt.

5.4. Fallstudie: DEC Alpha. Das Betriebssystem Digital Unix 4.0D enthält einen „class scheduler“. Hier können Prozesse und Prozessgruppen zu einer „Klasse“ zusammengefasst und dieser Klasse ein gewisser Anteil Rechenzeit zugeteilt werden. Das Betriebssystem versucht, jeden Prozess möglichst auf einem bestimmten Prozessor zu halten. Falls in einer Klasse nicht genügend Prozesse ablaufen können, um die zugeteilte Rechenzeit zu nutzen, wird diese Rechenzeit an andere Programme vergeben. Programme können ihren Ressourcenbedarf, inklusive der Zahl der benötigten Prozessoren, zur Laufzeit ändern.
Da parallele Anwendungen hier normalerweise nicht auf besondere Weise angemeldet oder gestartet werden, müssen für die Kommunikation mit dem Gang-Scheduler die Anwendungen selbst geändert werden. Eine Anwendung, die koordiniert ablaufen soll, muss einige spezielle Bibliotheksfunktionen ausführen:

- Registrieren des Programms beim Gang-Scheduler
- Angabe des Ressourcenbedarfs
- Angabe der zusammengehörenden Prozessgruppe

Diese Aufrufe könnten auch transparent in z.B. MPI- und PVM-Bibliotheken untergebracht werden, so dass Anwendungen, die diese Bibliotheken nutzen, nicht mehr selbst geändert werden müssten.

Der Gang-Scheduler von Digital Unix kann Programme über mehrere Rechner eines Clusters verteilen. Um das zu ermöglichen, benutzt der Scheduler eine feste Zeitscheibe. Die einzelnen Dämonen tauschen „Tickets“ mit Ressourcenangaben aus. Falls sich die Zahl der für einen Job benötigten Prozessoren ändert, werden zusätzliche Tickets eingeführt bzw. Tickets zurückgezogen. Dieser Gang-Scheduler erlaubt es auch, Prozessoren vom Gang-Scheduling auszunehmen, die dann für mit dem Benutzer interagierende Jobs zur Verfügung stehen.

5.5. Zusammenfassung der Fallstudien. Insgesamt wurde gezeigt, dass Gang-Scheduling sich gut für Multiprozessorsysteme eignet: mit dem Anwender interagierende Jobs und andere Jobs mit hoher Priorität können mit kurzer Antwortzeit und hohem Durchsatz ablaufen. Auch können Jobs mit großen Ressourcenanforderungen gestartet werden, ohne erst die Beendigung anderer Programme abwarten zu müssen. Über eine weite Spanne von unterschiedlichen Arbeitsbelastungen erzielt Gang-Scheduling eine gute Auslastung der Prozessoren.

6. Zusammenfassung

Gang-Scheduling kann bei Programmen mit feinkörniger Interaktion deutliche Verbesserung gegenüber reinem time-sharing bringen. Dynamisches Anpassen der Zahl der Prozesse kann eine weitere Verbesserung bewirken. Allerdings erreicht auch ein so modifiziertes Gang-Scheduling noch nicht die Leistung, die Space-sharing-Verfahren erreichen können. Beim Einsatz von Space-sharing-Verfahren kann allerdings die Ansprechbarkeit sehr beschränkt sein; damit sind sie für den Betrieb von mit dem Anwender interagierenden Programmen schlecht geeignet. Im praktischen Einsatz ist Gang-Scheduling manchen unkoordinierten Schedulern deutlich überlegen und liefert gute Auslastung bei gleichzeitig guter Interaktivität mit dem Anwender.

Literaturverzeichnis

[CL99] Julita Corbalán and Jesús Labarta. Dynamic speedup calculation through self-analysis. In *Technical Report number UPC-DAC-1999-43, Departament d'Arquitectura de Computadors, Universitat Politcnica de Catalunya*, 1999.

[CML01] Julita Corbalán, Xavier Martorell, and Jesús Labarta. Improving gang scheduling through job performance analysis and malleability. In *Proceedings of 15th Int. Conference on Supercomputing (SC2001)*, pages 303–311, 2001.

[FR92] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchonization. In *Journal of Parallel and Distributed Computing*, pages 306–318, 1992.

[Jet97] Morris A. Jette. Performance characteristics of gang scheduling in multiprogrammed environments. In *Proceedings of 11th Int. Conference on Supercomputing (SC97)*, November 1997.

[Ous82] John Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of 3rd Int. Conference on Distributed Computing Systems*, pages 22–30, October 1982.

KAPITEL 3

Novel Network Systems
Overlay Netzwerke & Peer-to-Peer Systeme

Seminarbeitrag von **Mike Sibler**

1. Einleitung

„Heute und für die nächsten Jahre lässt sich die wichtigste Entwicklung im Computerbereich mit Suns altem Slogan beschreiben: „Das Netzwerk ist der Computer.“ Wir bewegen uns weg von der Dominanz der Desktop-Systeme hin zu einer Welt allgegenwärtiger Rechnerdienste, eine Welt, in der Ihre wichtigsten Daten nicht mehr auf der lokalen Festplatte oder nicht einmal mehr in Ihrer Firmendatenbank gespeichert sind. In dieser neuen Welt werden Sie mit einer Vielfalt verschiedenster Geräte Informationen sammeln, Applikationen verwenden und Menschen kontaktieren und das alles, ohne notwendigerweise zu wissen, an welchem physischen Ort die involvierten Systeme und Geräte liegen.“ [**O'R**]

Tim O'Reilly

In Zeiten flächendeckender Verbreitung immer leistungsfähigerer Computersysteme und einem zunehmenden Grade der Vernetzung zwischen diesen, gelangen heutige Vernetzungstechnologien immer häufiger an ihre technischen Grenzen. Auch weisen neue Applikationen, wie z.B. Multimediaanwendungen, immer höhere Ansprüche an Netzwerkverbindungen auf. Da diese Bedürfnisse mit dem gegenwärtigen Stand der Netzwerktechnik nicht zu befriedigen sind, werden neue Netzwerktechniken, die unter dem Begriff Novel Network Technologies zusammengefasst werden, erforscht.

In der nun folgenden Ausarbeitung werde ich den Fokus auf zwei spezifische Technologiebereiche legen, die zweifellos den Novel Network Technologien zugeordnet werden können. Es handelt sich hierbei um Overlay Netzwerke und Peer-to-Peer (P2P) Systeme.

Overlay Netzwerke stellen eine der physikalischen Netzwerkschicht übergeordnete logische Schicht dar, die in den hier vorgestellten Fällen der Realisierung einer räumlichen Struktur (Topologie) der Daten innerhalb des Netzwerks dient. Hierbei

kann auf der Ebene des Overlays ein beliebiger Abstraktionsgrad von den (physikalischen) Verbindungen der Knoten auf darunterliegenden Schichten erzielt werden.

Um dem Leser den Grundgedanken von Peer-to-Peer Systemen näherzubringen, möchte ich David Gelerter, Professor an der Universität von Yale, zitieren:

„If a million people use a Web site simultaneously, doesn't that mean that we must have a heavy-duty remote server to keep them all happy? No; we could move the site onto a million desktops and use the internet for coordination. [**Gel**]

David Gelerter

Bei Peer-to-Peer Systeme kommen also ungenutzt Ressourcen weitverbreiteter Desktop-Computer zum Einsatz. Wichtig ist hierbei, dass nach der strenge Definition von P2P-Technologien die Knoten des P2P-Netzes - die so genannten Peers - völlig gleichberechtigt und dezentral organisiert sind, d.h. es sind keinerlei zentrale Kontroll- und Steuereinheiten für den Betrieb eines P2P-Netzes vorgesehen.

In der nun folgenden Ausarbeitung werde ich die wichtigsten Konzepte obiger Technologien darstellen und diese anhand repräsentativer Anwendungen genauer betrachten.

2. Overlay Netzwerke

2.1. Was ist ein Overlay Netzwerk? Overlay Netzwerke sind bei weitem keine neue Technologie. Bereits das Internet selbst wurde als Overlay des Telefonnetzwerkes entwickelt.

Overlay Netzwerke sind Netzwerke, die auf ein gegebenes physikalisches Netzwerk, welches die realen physikalischen Verbindungen zwischen den Knoten realisiert, aufgesetzt werden. Es handelt sich somit, wie der Name bereits sagt, um eine der physikalischen Netzwerkschicht übergeordnete Schicht.

In Overlay Netzwerk wird also von der physikalischen Netzwerkschicht abstrahiert und die Netzwerkstruktur auf einer höheren, logischen Ebene betrachtet. Die Topologie solcher Overlay Netzwerke kann hierbei je nach Verwendungszweck hochdynamische und äußerst komplexe Gestalt annehmen. Grundsätzlich gilt, dass ein Overlay Netzwerk in keinerlei topologischen Zusammenhang mit dem zugrunde liegenden physikalischen Netzwerk stehen muss.

Aufgrund der sehr allgemeinen Definition von Overlay Netzwerken sind die Einsatzgebiete entsprechend breit gefächert. Um das Prinzip von Overlay Netzwerken zu verstehen, werde ich im folgenden zwei typische Anwendungen, die Gegenstand aktueller Forschungsprojekte sind, vorstellen.

2.2. Resilient Overlay Netzwerke (RON)[ABKM02]. Ein Resilient (dt.: wiederstandsfähiges) Overlay Netzwerk ist eine Overlay Technologie, die verteilten Internetanwendungen das Erkennen von Pfadausfällen und deren Behebung durch Rerouting innerhalb weniger Sekunden ermöglichen soll. Damit stellen RON Netzwerke einen Mechanismus zur Erhöhung der Robustheit heutiger Weitverkehrsnetze dar, da diese meist mehrere Minuten für ein Rerouting im Falle eines Pfadausfalls benötigen. RON ist realisiert als eine Applikationsschicht Overlay, das auf die vorhandenen Internetschichten aufgesetzt wird.

Um den Ansatz von RONs besser verstehen zu können, werde ich zunächst einen kurzen Exkurs über den Aufbau der Internet Architektur geben. Das heutige Internet ist aus unabhängig agierenden Autonomen Systemen (ASs) zusammengesetzt, die über verschiedene Pfade direkt oder indirekt miteinander verbunden sind

(teilvermaschtes Netz). Detaillierte Routinginformationen werden hierbei nur in den Routingtabellen einzelner AS und dem dazugehörigen Netz, welches üblicherweise von einem Internet Service Provider (ISP) betrieben wird, verwaltet. Diese Routinginformationen werden in Grenzroutern, die zur Vermittlung von Paketen zwischen jeweils benachbarten Netzen eingesetzt werden, unter Verwendung des Border Gateway Protokolls (BGP-4) stark gefiltert, so dass diese im Nachbarnetz nicht mehr zur Verfügung stehen. Hierdurch erhöht sich die Skalierbarkeit des Internets, da der Datenaustausch zwischen Teilnetzen auf ein Minimum reduziert wird. Die Fehlertoleranz der Ende-zu-Ende Kommunikation wird jedoch stark reduziert, da im allgemeinen keine Aussage über den Zustand benachbarter Netzwerke gemacht werden kann. Dies führt dazu, dass Pakete ohne Berücksichtigung der Netzlast und ähnlicher Parameter, die für die Ende-zu-Ende Zuverlässigkeit entscheidend sind, geroutet werden. Fällt nun ein Pfad aus oder kommt es zu gravierenden Leistungseinbußen auf einem solchen, steigt die durchschnittliche Paketverlustrate und der Datendurchsatz sinkt, was bei den meisten Protokollen, darunter auch TCP, zu einem starken Leistungsabfall bzw. einem Totalausfall führt.

Entwurfsziele von RONs. Resilient Overlay Netzwerke erheben den Anspruch einige der oben angesprochenen Probleme der heutigen Internetarchitektur beheben zu können. Hierbei wurden beim Entwurf von RON der Schwerpunkt auf drei Hauptziele gelegt: Schnelles Erkennen von Ausfällen und Wiederherstellung von Verbindung in einer Gesamtzeit von weniger als 20 Sekunden; stärkere Integration des Routings und der Pfadwahl in die jeweilige Applikation und die Erstellung eines Frameworks zur Implementierung mächtiger Routing Policies.

(1) **Schnelles Erkennen von Ausfällen/Wiederherstellung von Verbindungen** Die Knoten eines RON Netzwerkes haben die Möglichkeit Information über die Güte der sie verbindenden Pfade auszutauschen. Diese Informationen werden verwendet, um im Falle von Problemen mit dem Internetpfad eine alternative Verbindungroute zu finden. Zu diesem Zweck bilden die Knoten des RONs ein kooperierendes Netzwerk. Sie überwachen die Funktion und Qualität der Internetpfade zwischen sich und ihren jeweiligen Partnern. Diese Information verwenden sie, um darüber zu entscheiden, ob die Pakete direkt über das Internet geleitet werden sollen oder über einen alternativen Pfad, der durch Umleitung der Pakete an benachbarte RON-Knoten aufgebaut wird. Hierbei hat sich in der Praxis gezeigt, dass Umleitungen um fehlerhafte Pfade meist durch nur einen zusätzlichen Hop im RON-Netzwerk realisiert werden können.

    Die Knoten dieser Resilient Overlay Netzwerkes sind normalerweise auf eine Vielzahl verschiedener Routingdomains verteilt, so dass die Wahrscheinlichkeit eines Ausfalls sämtlicher Knoten eine RON Netzwerkes als realtiv klein eingeschätzt werden kann, wodurch die Robustheit eines RON als sehr hoch eingeschätzt wird.

    RON Knoten tauschen Informationen über die Güte der Pfade durch ein spezielles Routingprotokoll aus und erstellen anhand dieser Informationen Routingtabellen, die eine Vielzahl von Pfadparametern, wie z.B. Verzögerung, Paketverlustrate und Durchsatz der jeweiligen Verbindungen enthalten. Hierzu führt jeder RON Knoten Messungen und Bewertungen des ankommenden Datenverkehrs durch. Damit der Overhead, der durch diese Messungen erzeugt wird, nicht übermäßig groß wird, sollten RON Netzwerke nur etwa 2 bis 50 Knoten umfassen.

(2) **Integration des Routings in Applikationen** Die Reaktion auf Störungen in Internetverbindungen variieren von Anwendung zu Anwendung. Netzwerkbedingungen, die für eine Anwendung fatale Folgen haben,

können für eine andere durchaus akzeptabel sein. Ein Beispiel hierfür: Während für eine TCP-Anwendung eine durchschnittliche Paketverlustrate von 10%, aufgrund spezieller Adaptionsmechanismen, nur mit Leistungseinbußen einhergeht, bedeutet diese für ein auf UDP basiertes Internetradio mit ungenügender Fehlerbehandlung die absolute Nichtverwendbarkeit, da das Audiosignal ständig abreißt.

Um den Applikationen die Möglichkeit zu geben, die für sie wichtigen Pfadparameter individuell beeinflussen zu können, werden Routing und Pfadselektion in RON-Systemen stärker in die entsprechenden Anwendungen integriert. Jede Applikation kann hierzu entsprechenden Pfadparameter priorisieren, z.B. geringe Latenz wichtiger als hoher Durchsatz oder geringer Paketverlust wichtiger als geringe Wartezeit. Klar ist jedoch, dass ein Routingsystem nicht in der Lage ist alle diese Parameter gleichzeitig zu optimieren. Erste Versionen der RON Implementierung erlauben nur eine Optimierung bzgl. eines Pfadparameters.

Aufgrund dieser anwendungsspezifischen Anpassungsfähigkeit können RONs für eine Vielzahl von Anwendungen verwendet werden. Eines der besten Beispiele ist der Einsatz in einer Applikation für Multimedia-Konferenzen. Die Anwendung baut hierzu ein Resilient Overlay Network zwischen den Konferenzteilnehmern auf und wählt die Verbindungspfade zwischen diesen bzgl. Parametern wie Verlustrate, Laufzeitverzögerungen oder Durchsatz individuell und optimal.

(3) **Framework für Policy Routing** RONs bieten auch ein Framework zur Implementierung von Policy Routing an. Policy-Routing beeinflusst ebenfalls die Auswahl von Pfaden im Netzwerk, wobei hier jedoch weniger die technischen Parameter des Pfades eine Rolle spielen sondern so genannte strategische Entscheidungen. Ein Beispiel einer solchen strategischen Entscheidung wäre dass spezielle Pfade, die über eine sehr hohe Bandbreite verfügen, nur von Benutzern verwendet werden, die diese Mehrleistung auch bezahlt haben. Heutige Ansätze des Policy Routings sind meist sehr primitiv und umständlich. So ist z.B. mit BGP-4 die Definition von feingranularen Policies auf Benutzerbasis nicht möglich sondern wird nur anhand der Quell- und Zieladresse des Pakets durchgeführt.

Zur Realisierung des Policy Routings wird jedes Paket innerhalb eines RON unter Verwendung eines Datenklassifizieres klassifiziert und um ein entsprechendes Policy-Tag erweitert. Dieses Tag bestimmt im jeweiligen RON-Router eine entsprechende Menge von Routingtabellen, die für die Weiterleitung dieses Paktes verwendet werden können. Für jede Policy wird in jedem RON-Router eine Menge von Routingtabellen verwaltet, wobei durch ständige Berechnungen die nicht erlaubten Routingpfade der jeweiligen Policy aus der entsprechenden Tabelle entfernt werden. Diese Routingberechnungen berechnen für die entsprechende Policy den kürzesten Pfad von einem RON-Knoten zu jedem anderen Knoten innerhalb dieses Rons (max. 50 Knoten).

Da RONs für den Einsatz auf Endsystemen vorgesehen werden, die meist sehr leistungsfähig sind und durch eine Vielzahl redundanter Pfade miteinander verbunden sind, sind nach Ansicht der Architekten der RON die technischen Voraussetzungen für die Realisierung dieses feingranularen Policy-Routings auf Benutzerbasis gegeben.

Architektur eine RONs. Wie bereits zuvor erwähnt, bilden RON-Knoten eine kooperierende Schicht auf Anwendungsebene mit der Aufgabe des gegenseitigen

Routings von Paketen. Die Architektur der RONs, die ein Routing auf der Grundlage verschiedener Routingparameter ermöglicht, zeichnet sich durch einen einfachen Aufbau aus. Abb. 1 stellt diesen Aufbau schematisch dar.

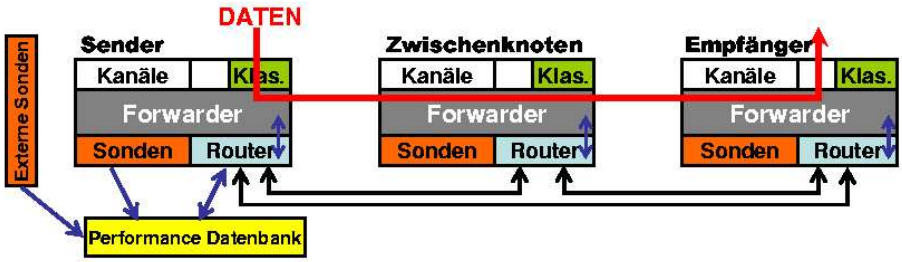


ABBILDUNG 1. Die Architektur eines RON-Systems

Die Datenpakete werden hierbei von RON-Klienten über einen Kanal (engl: conduit) an einen Eingangsknoten geleitet. In jedem Knoten verwendet der Forwarder die Informationen des Routers, um den besten Pfad zu finden und das Paket in die entsprechende Richtung weiterzuleiten. Der Router greift hierbei auf Informationen der Performance Datenbank zurück, in der die Ergebnisse der innerhalb eines RON-Netzwerks durchgeführten Messungen abgelegt sind. Diese Messungen bewerten die Güte einer Verbindung bzgl. verschiedener Parameter. Der Router hat also die Aufgabe anhand der Informationen in den Routingtabellen, den kürzesten, anwendungsoptimalen und policygerechten Weg durchs Netz zu finden. Der Eingangssknoten ist in diesem Prozess für die Festlegung des kompletten Pfades eines Pakets durch das Netz und die Vergabe des Policy-Tags zuständig und teilt sämtliche Informationen, die für die Weiterleitung des Paketes notwendig sind, allen Knoten auf dem entsprechenden Pfad durch ein Update der Routingtabellen mit. Alle Zwischenknoten müssen folglich ein entsprechendes Paket nur noch auf dem vorbestimmten Weg durchs Netz weiterleiten und keine Neuberechnungen durchführen. Angekommen beim Empfänger, wird das Paket vom Forwarder an den entsprechenden Ausgangskanal weitergeleite und die entsprechenden Policy Tags entfernt.

Erste Erfahrungswerte von RONs in der Praxis. Seit Anfang 2001 befinden sich erste Implementierungen von RONs im Testbetrieb. In einem RON-Netz, bestehend aus 16 Knoten verteilt über die ganze Welt, hat sich in Messungen gezeigt, dass RON in der Lage war zwischen 60% und 100% aller Ausfälle durch Rerouting zu umgehen. Die Implementierungen benötigten hierbei im Durchschnitt 18 Sekunden für die Erkennung von Pfadausfällen und deren Behebung durch Aufbau von alternativen Wegen und waren somit wesentlich schneller als die üblichen Protokolle der heutigen Internetarchitektur.

Darüber hinaus war RON in der Lage auch Leistungsabfälle auf Pfaden zu umgehen. Bei 5% aller Messungen wurde die Paketverlustrate um mehr als 5% verrigngert, die durchschnittliche Ende-zu-Ende Wartezeit wurde in 11% aller Fälle um mehr als 40 ms reduziert und der TCP-Durchsatz verdoppelte sich in mehr als 50% aller Fälle.

Diese gemessenen Verbesserungen, besonders im Bereich der Ausfallkontrolle, demonstrieren die Vorzüge, die sich durch eine Verschiebung von Teilen der Wegewahl auf Anwendungsebene in den Endsystemen ergeben können.

2.3. Semantische Overlay Netzwerke für P2P Systeme. Semantischen Overlay Netzwerke (SON) wurden entworfen, um der schlechte Skalierbarkeit und der Ineffizienz von Suchanfragen gegenwärtiger P2P-Systeme entgegenzuwirken. In unstrukturierten Systemen, wie diese heutzutage in der Welt des Internets allgegenwärtig sind, werden Suchanfragen „blind“, d.h. auf zufällige Art und Weise, durch das Netzwerk geleitet.

SONs verfolgen nun den Ansatz, dass Knoten mit semantisch ähnlichem Inhalt in einem Cluster gebündelt werden. Diesen Sachverhalt am Beispiel einer Musiktauschbörse wird in Abb. 2 dargestellt.

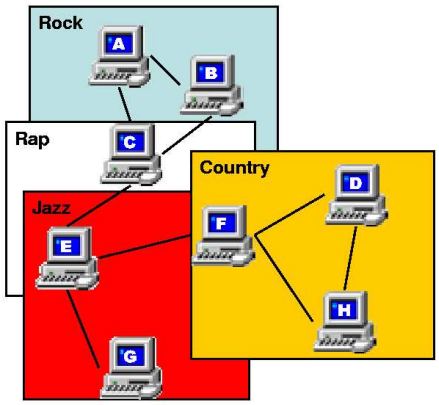


Abbildung 2. Schema eines semantischen P2P Overlay Netzwerkes

Die durchgezogenen Linien zwischen den Knoten A bis H stellen hierbei die physikalischen Verbindungen zwischen diesen Knoten dar. Bei SONs handelt es sich um den Zusammenschluss von Knoten mit ähnlichem Inhalt (hier in Form von Rechtecken dargestellt). So enthalten z.B. die Knoten A, B und C Rocksongs, so dass auf dem Niveau des semantischen Overlay Netzwerks eine logische Beziehung zwischen diesen drei Knoten besteht. Hierbei ist jedoch wichtig, dass Beziehungen auf logischer Ebene nicht auf physikalischer Ebene abgebildet sein müssen, d.h. im SON Rap muss der Knoten F nicht direkt mit Knoten C verbunden sein, obwohl diese innerhalb ihres SONs in direkter Beziehung in direkter Beziehung zueinander stehen. Außerdem kann es auch Knoten geben, die mehr als einem SON angehören, wie der Knoten F in obiger Abbildung. Darüber hinaus können SONs hierarchisch gegliedert sein, so dass z.B. der SON Rock Sub-SONs, wie z.B. Softrock und Hardrock, enthalten könnte.

Wird nun eine Suchanfrage an ein Netzwerk mit semantischen Overlay gestellt, wird im ersten Schritt festgestellt welcher SON für die Beantwortung der Anfrage am Besten geeignet ist. Wurde der SON festgelegt, wird die Anfrage nur noch an die Knoten weitergeleitet, die dem entsprechenden SON angehören. Alle anderen Knoten bleiben unberührt. Hierdurch wird die Bearbeitungszeit der Anfrage reduziert, da weniger Knoten untersucht werden müssen, und die nicht untersuchten Knoten können in der Zwischenzeit bereits eine andere Anfrage bearbeiten.

Natürlich enthält dieser Ansatz eine Menge Herausforderungen, wie z.B. die Klassifikation von Knoten, d.h. eine Antwort auf die Frage: Was bedeutet *„ein Knoten der Rocksongs enthält“*? Auch muss die Granularität der Klassifikation definiert werden, da eine zu detaillierte Unterteilung die administrativen Kosten erhöht, während eine zu grobe Klassifizierung nicht genügend Lokalität bereitstellen würde.

Nicht zuletzt muss ein Mechanismus zur Verfügung gestellt werden, der zu einer gegebenen Anfrage den richtigen SON auswählt, der z.B. einen Song von Miles Davis dem SON Jazz zuordnet.

Für genauere Details der Realisierung semantischer Overlay Netzwerke, sei auf [**CGM**] verwiesen.

3. Peer to Peer (P2P)

3.1. Was bedeutet P2P?. Spätestens seit dem heftigen Streit um Copyrights zwischen der Musiktauschbörse Napster und der RIAA[1] im Jahre 2001 ist der Ausdruck Peer-to-Peer selbst computertechnisch unversierten Leuten ein Begriff. Die meisten Personen verbinden diesen Ausdruck jedoch nur mit der Technik zur illegalen Verbreitung von Copyright geschütztem Material. Doch hinter P2P verbirgt sich weit mehr.

Doch was ist P2P nun eigentlich? Für den Ausdruck „Peer-to-Peer“ gibt es keine offizielle formale Definition. Peer bedeutet in der deutschen Sprache gleichrangig/ebenbürtig. Übertragen auf Netzwerktechnologien bedeutet P2P somit eine direkte Verbindung zwischen gleichrangigen Systemen. Dies deckt sich mit der Definition der Peer-to-Peer Working-Group[2]:

„Peer-to-Peer computing is sharing of computer resources and services by direct exchange“.

Bei P2P verläuft die Kommunikation also direkt zwischen den Klienten und nicht, wie diese bei der typischen Client-Server Kommunikation der Fall ist, über eine zentrale Kontroll- und Steuerinstanz.

Eine weitere Definition von P2P ist nach Clay Shirky[**Shi00**]:

„P2P is a class of applications that takes advantage of resources - storage, cycles, content, human presence - available at the edges of the Internet.“

Clay Shirky

Der Zugriff auf solche dezentralisierte Ressourcen an den Rändern des Internets (d.h. den Endanwendersystemen) impliziert jedoch, dass die Arbeitsumgebung von P2P Systeme durch instabile Verbindungen und unvorhersehbare Adressen der Teilnehmer geprägt ist. P2P Knoten arbeiten deshalb außerhalb des DNS (Domain Name Service) Systems und sind somit im Allgemeinen von zentralen Servern unabhängig.

Da sich die Topologie von P2P-Netzwerken, aufgrund von Knoten die das Netz verlassen oder neu hinzukommen, stets verändert, ist ein Mechanismus zur Selbstorganisation der Netzwerkstruktur für eine P2P-System ein absolutes Muss. Somit sind die Herausforderungen an die Organisation eines P2P Netzwerke, verglichen mit denen eines Standard Client/Server Netzwerk, ungleich höher. Will man wissen, ob ein gegebenes System als P2P eingestuft werden kann, so kann man nach [**Shi00**] folgende Faustregeln zu Rate ziehen:

- Kann das System mit instabilen Netzwerkverbindungen und temporären Netzwerkadressen umgehen?
- Sind die Knoten am Rande des Netzwerks weitestgehend autonom?

Nur wenn beide Fragen mit ja beantwortet werden können, handelt es sich um ein P2P-System im klassischen Sinne.

Obwohl P2P in den letzten Monaten immer wieder als zukunftsweisendes Konzept in Erscheinung trat, handelt es sich dabei keinesfalls um neue Technologie. Bereits ältere Generationen von Servern, wie z.B. News, Email und IRC-Server[3], realisieren den Austausch von Nachrichten auf P2P-Basis.

Obwohl P2P heute von der breiten Masse der Anwender zum File Sharing verwendet wird, ist dies nur eines der möglichen Anwendungsgebiete. P2P-Technologien werden darüber hinaus für Applikationen zum Distributed Processing, Instant Messaging und Collaborative Computing eingesetzt.

---

[1] Recording Industry Association of America

[2] Zusammenschluß von HP, Intel, IBM,... mit dem Ziel die Infrastruktur von P2P Systemen zu verbessern

[3] Internet Relay Chat

Die folgenden Kapitel werden einen Überblick über die oben genannten Anwendungsgebiete von P2P geben und deren technische Details genauer betrachten. Die Abgrenzungen zwischen den verschiedenen Typen von Anwendungen sind dabei fließend, da verschiedenen Funktionalitäten oftmals in einer Anwendung kombiniert sind.

3.2. File Sharing. Wie zuvor bereits erwähnt, ist File Sharing die Anwendung, die von den meisten Usern mit dem Begriff P2P in Verbindung gebracht wird. File Sharing umfasst in diesem Kontext Systeme, die den direkten Austausch von Daten (z.B. Musik, Video, Dokumenten, etc.) zwischen Klientensystemen unterstützen. Beispiele solcher Anwendungen sind Napster, Gnutella, KaZaA, eDonkey, Audio Galaxy und viele mehr.

***Klassifizierung von P2P File Sharing Architekturen*[AT]**. P2P File Sharing Architekturen zeichnen sich durch zwei grundlegende Aspekte aus, die im Folgenden der Klassifikation dienen: der Grad der Zentralisierung des Netzwerkes und die realisierte Netzwerkstruktur.

(1) **Unterteilung nach dem Grad der Zentralisierung** Hierbei werden P2P File Sharing Anwendungen anhand des Grades der Zentralisierung der Knoten bewertet, d.h. man bewertet den Grad der Verwendung zentraler Einheiten, die zur Realisierung der Interaktion zwischen Peers eingesetzt werden.

Man unterscheidet drei Zentralisierungsebenen bei File Sharing Systemen:

- *Komplett dezentralisiert:* Alle Knoten im Netzwerk sind gleichberechtigt und führen die gleichen Aufgaben aus, jeder Knoten agiert sowohl als Server und Klient, weshalb die Knoten solcher Netze auch als „Servents“ (SERVers+cliENTS) bezeichnet werden. Es gibt keinerlei zentrale Koordination der Knoten.
- *Teilweise dezentralisiert:* Teilweise dezentralisierte Systeme bauen auf der Architektur von komplett dezentralisierten Systemen auf. Es gibt jedoch einige Knoten, die eine wichtigere Rolle als die restlichen haben. Diese Knoten - auch Superknoten genannt - dienen als lokaler, zentraler Index für Dateien, die von den lokalen Peers geshared werden. Die Art und Weise wie Knoten die Rolle von Superknoten zugewiesen werden, variiert von System zu System. Wichtig hierbei ist jedoch dass diese Superknoten keine zentralen Ausfallpunkte für das P2P-System darstellen dürfen, d.h. dass diese Knoten vom System dynamisch durch andere ersetzt werden können müssen.
- *Hybrid dezentralisiert:* In diesen Systemen gibt es einen zentralen Server, der die Interaktion zwischen den Peers gewährleistet indem er Verzeichnisse bereitstellt, die die freigegebenen Dateien aller Knoten des Netzwerks, in Form von Metadaten, verwaltet. Die Ende-zu-Ende Interaktion (der eigentliche Download) zwischen 2 Peers findet jedoch direkt zwischen den Peers statt. Der zentrale Server dient in solchen Systemen nur dem Auffinden von Dateien und der Identifikation von Knoten, die die gesuchten Dateien enthalten. Damit entspricht diese Architektur weitestgehend dem Prinzip des Client/Server Paradigmas und nur der Filetransfer findet direkt zwischen den Peers statt. Folglich haben diese Systeme einen zentralen Punkt, was diese Architektur anfällig gegenüber Zensuren, technischen Schwierigkeiten oder aber Angriffen macht. Solche Systeme werden im Allgemeinen

nicht als reine P2P-Systeme betrachtet, man spricht stattdessen von „*Peer-through-Peer*“ Systemen.

(2) **Netzwerkstruktur** Unter der Netzwerkstruktur versteht man die räumliche Struktur - die Topologie - des Netzwerkes welche dem Datenaustausch zu Grunde liegt. Dabei wird vom physikalischen Netzwerk, d.h. der direkten physikalischen Verbindung der Klienten untereinander, abstrahiert und die Netzwerkstruktur auf einer höheren, logischen Ebene (meist inhaltliche Zusammenhänge zwischen Dateien) betrachtet. Es handelt somit bei dieser logischen Ebene um ein Overlay des P2P-Systems.

P2P Systeme können anhand der Struktur ihres verwendeten Overlay-Netzwerks klassifiziert werden. Will man ein P2P System in eine Strukturklasse einteilen, muss man sich die folgende Frage stellen: Ist von vornherein bekannt, auf welchen Knoten ein spezieller Inhalt zu finden ist oder müssen wir zufallsbasiert das Netzwerk nach Inhalten durchsuchen?

- *Unstrukturierte Netzwerke:* Der Ort, an dem die Daten abgelegt wurden, ist völlig unabhängig von evtl. vorhandenen Beziehungen der Dateien des P2P Systems. Da man für eine gesuchte Datei somit keinerlei Informationen über die Knoten hat, die dieses speichern, muss das Netzwerk „auf gut Glück“ durchsucht werden. Es werden also beliebige Knoten betrachtet und geprüft ob diese die gesuchte Datei enthalten.
  Unstrukturierte Netzwerke haben den Vorteil, dass sie eine große Anzahl an Knoten umfassen können, da die Organisationsstruktur sehr einfach ist. Das Problem ist jedoch die mangelnde Skalierbarkeit der unstrukturierten Netzwerke, da es mit zunehmender Teilnehmerzahl immer schwieriger wird, eine gesuchte Datei zu finden, da die Wege und somit die Anzahl der betrachteten Knoten im Netz immer länger werden.
- *Strukturierte Netzwerke:* Strukturierte Netzwerke wurden entworfen, um die Skalierbarkeitsprobleme der unstrukturierten Systeme zu lösen. Bei strukturierten Netzwerken hat man eine zusätzliche Kontrolle auf der Ebene des Overlay Netzwerks, welche dafür sorgt, dass die Dateien (oder Zeiger auf diese) nur an genau festgelegten Positionen des Netzwerkes abgelegt werden können. Diese zusätzliche Kontrolle wird erreicht, in dem man verteilte Routingtabellen hinzufügt, die eine Zuordnung von Dateiidentifikator auf Ablageort der Dateien realisieren. Somit können Anfragen auf effektive Art auf direktem Weg zum Knoten, der die gesuchte Datei enthält, geleitet werden.
  Diese Systeme zeichnen sich somit durch eine bessere Skalierbarkeit für exakte Anfragen aus. Leider sind solche Routingmechanismen in einer variablen Systemumgebung, in der Knoten sich jederzeit im Netzwerk ab- oder anmelden können, nur sehr schwer effizient implementierbar.
- *Lose Strukturierte Netzwerke:* Hierbei handelt es sich um eine Mischung der beiden obigen Netzwerktypen. Der Ablageort der Datei wird durch Routinghinweise beeinflusst, aber das Routing ist nicht komplett spezifiziert, so dass nicht alle Suchanfragen erfolgreich ausgeführt werden können.

Übersicht über P2P File Sharing Anwendungen. Die folgende Übersicht ordnet heutige P2P File Sharing Anwendungen in die zuvor vorgestellten Klassifizierung ein. Selbstverständlich handelt es sich hierbei nicht um eine vollständige Aufzählung aller P2P Systeme, aber die hier genannten Vertreter sind aufgrund ihrer Architektur für die heute im Einsatz befindlichen P2P Systeme repräsentativ.

	Unstruktukurierte Netzwerke	Lose Strukturierte Netzwerke	Strukturierte Netzwerke
Hybrid dezentralisiert	Napster		
Komplett dezentralisiert	Gnutella	Freenet	Chord,Pastry
Teilweise dezentralisiert	KaZaA,Gnutella		

In den folgenden Unterkapiteln werde ich nun die einzelnen, in der obigen Übersicht genannten Vertreter der jeweiligen Klassen und deren zugrunde liegende Technik kurz vorstellen.

Unstrukturierte P2P-Systeme.

Hybride dezentralisierte unstrukturierte Systeme. Bekanntester Vertreter dieser Klasse ist Napster, das als erste Massentauschbörse und späteren Gerichtsprozessen gegen die Plattenindustrie berüchtigtem Ruhm erlangte. Abb. 3 stellt die Architektur Napsters und den Verlauf eines Such- und Downloadvorganges im Napster Netzwerk dar.

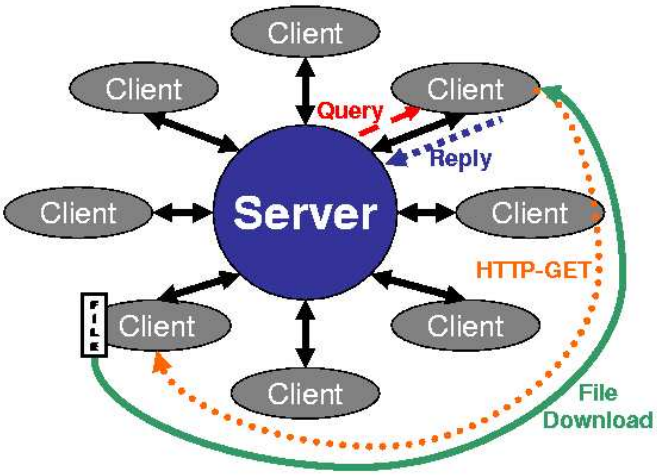


ABBILDUNG 3. Schematische Darstellung von Query, Reply und Dateidownload bei Napster

Aufgrund des zentralen Servers, handelt es sich bei der Architektur von Napster nicht um eine P2P Applikation im engeren Sinne, sondern, wie bereits zuvor erwähnt, um eine Peer-trough-Peer Applikation. Napster ist realisiert als ein Netzwerk von registrierten Nutzern, die eine Client-Software lokal installiert haben. Der zentrale Verzeichnisserver ist zuständig für:

- Indizierung aller im Netzwerke vorhandenen Dateien durch Metadaten (Dateiname, Erstellungszeitpunkt, etc.
- Bereitstellung einer Tabelle mit Verbindungsinformationen der registrierten Benutzer (IP-Adresse, Übertragungsgeschwindigkeit, etc.)
- Bereitstellung einer Tabelle die für jeden Benutzer die von ihm freigegebenen Dateien verwaltet

Beim Login des Benutzers ins Napster System wird dieser mit dem zentralen Server verbunden und übergibt eine Liste all seiner freigegebenen Dateien. Erhält der Server nun eine Anfrage nach einer bestimmten Datei, sucht er nach Übereinstimmungen in seiner Datenbank und gibt eine Liste von Benutzern, die diese Datei auf ihrem Rechner haben zurück. Will der Benutzer die Datei nun von einem anderen Knoten downloaden, baut er unter Verwendung eines standardisierten Http-Get Requests eine direkte Verbindung zu dem entsprechenden Peer auf. Der Knoten der die entsprechende Datei enthält, agiert nun als einfacher Webserver, der diese Anfrage beantwortet und die Datei direkt an den suchenden Knoten übermittelt. Man nennt einen solchen Vorgang eine Out-of-Network Verbindung, da die Pakete zwischen den Knoten nicht über die Netzinfrastruktur sondern direkt versendet werden.

Der Vorteil dieses und ähnlicher Systeme ist, dass sie sehr einfach aufgebaut sind und die Suche von Dateien schnell und effektiv vonstatten geht. Nachteilig ist jedoch die Verwundbarkeit des Systems gegenüber Attacken, Zensuren und technisch bedingten Ausfällen aufgrund der zentralen Architektur.

Komplett dezentralisierte unstrukturierte Systeme. Das Gnutella Netzwerk, dessen Ursprung bei Nullsoft (einer Tochtergesellschaft von America on Line (AOL)) liegt, weist wohl die interessanteste P2P Architektur auf, da diese auf einem völlig dezentralisierten und selbstorganisierenden Ansatz beruht.

In einem völlig dezentralisierten Netzwerk gibt es keine zentrale Koordinationsstelle und die Kommunikation zwischen teilnehmenden Knoten erfolgt direkt. In solchen Systemen wird eine bereits zuvor erwähnte Servent Software, die als Client und Server dient, verwendet. Eine Instanz dieser Servent Software auf einem Benutzersystem wird auch als „Host“ bezeichnet. Die folgende Abb. 4 stellt eine Momentaufnahme des Gnutella Netzwerkes vom Januar 2000 dar. Jeder Punkt repräsentiert hierbei einen Servent und jede Linie eine Netzwerkverbindung. Zum Zeitpunkt dieser Momentaufnahme waren etwa 31.000 Hosts mit einem Datenvolumen von 519 Terrabytes aktiv.

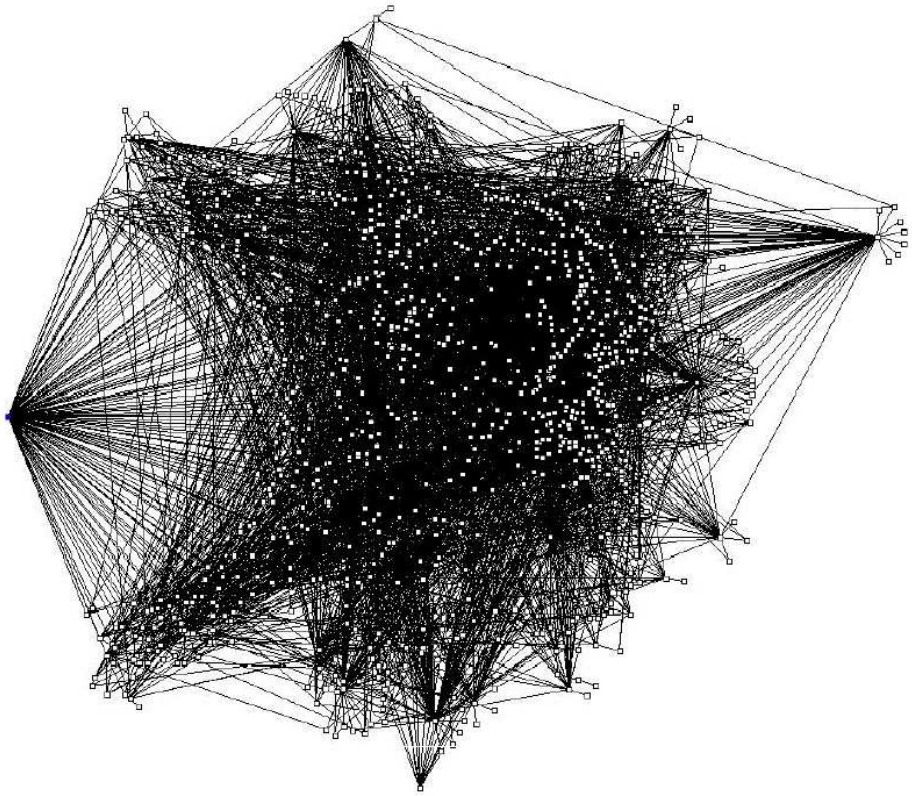


Abbildung 4. Momentaufnahme des Gnutella Netzwerks am 27.01.2000 [**Jov00**]

Um die Funktionsweise des Gnutella Systems zu erklären, muss man zwischen der Anmeldung an das Netzwerk, der Suche nach Dateien und dem eigentlichen Download unterscheiden.

Die Kommunikation zwischen Hosts ist als Protokoll auf Anwendungsebene spezifiziert, welches 4 Arten von Nachrichten umfasst.

- **Ping:** Anmeldungs-Anfrage eines bestimmten Hosts
- **Pong:** Antwort auf eine Ping Nachricht, enthält IP, Port, die Anzahl und Größe der freigegebenen Dateien des antwortenden Hosts
- **Query:** Eine Suchanfrage, welche einen Suchstring und die minimalen Geschwindigkeit des antwortenden Hosts enthält

- **QueryHit:** Antwort auf eine Suchanfrage. Sie enthält die IP, Port, Geschwindigkeit, die Anzahl der Übereinstimmungen und die URL der übereinstimmenden Daeien des antwortenden Hosts

Bei der Anmeldung ans Gnutella Netz wird eine Ping-Message zu allen direkt verbundenen Nachbarknoten mittels Broadcasting gesendet. Alle Nachbarn, die diese Ping-Nachricht erhalten, antworten mit einer Pong-Nachricht, in der sie sich selbst identifizieren und leiten die ursprüngliche Ping-Nachricht weiter an ihre Nachbarn. Diese Ping/Pong-Signalisierung wird auch während des Betriebs stetig wiederholt, um neu hinzugekommene oder abgemeldete Nachbarhosts zu erkennen und somit die selbstorganisierende Eigenschaft des Netzes zu gewährleisten.

Damit die Pong-Antwort auch den Weg zum Absender der Ping Nachricht wieder findet, wird jede Nachricht mit einer eindeutigen Nachrichten-ID gekennzeichnet. Jeder Host ist mit einer dynamischen Routingtabelle von Nachrichten-IDs ausgestattet. Die Antwort verwendet stets den gleichen Weg durchs Netz wie die Anfrage. Dies wird erreicht, indem die Antwort mit der gleichen Nachrichten-ID wie die Anfrage versehen wird. Kommt die Antwort bei einem Host an, so vergleicht dieser die Nachrichten-IDs in seiner Routingtabelle und findet den nächsten Hop auf dem Weg zum Ursprungsknoten der Ping-Nachricht. Hierbei ergibt sich jedoch das Problem, dass aufgrund der dynamischen Struktur des Netzes Knoten die auf dem Hinweg noch gültig waren, sich auf dem Rückweg abgemeldet haben können, wodurch eine entsprechende Pong-Nachricht verloren geht.

Diese eindeutige Nachrichten-ID wird zur Effizienzsteigerung auch verwendet um Schleifen zu verhindern, indem doppelt ankommende Nachrichten in einem Knoten verworfen werden (vgl. Abb. 5).

Um die durch Verwendung von Broadcasting nicht unerhebliche Verkehrslast im Netzwerk zu beschränken, verwendet man in Gnutella das TTL (Time To Live) Konzept. Hierbei handelt es sich um einen einfachen Zähler im Nachrichtenkopf, der bei jedem Hop im Netz heruntergezählt wird. Erreicht der Zähler den Wert 0 wird die Nachricht verworfen. TTL beschränkt somit die Länge der Pfade, die ein Paket im Netz zurücklegen kann. Somit kommuniziert ein Host also nur mit einer gewissen Untermenge aller Knoten im Netz, man spricht hierbei auch vom „Horizont“ des Hosts. Alle Dateien die hinter dem Horizont des Hosts liegen, sind für diesen sowohl unsichtbar als auch nicht zugreifbar. Das Gnutella Netzwerk ist daher aus Anwendersicht in Teilbereiche fragmentiert.

In Gnutella Netzwerken verwendet man typischerweise einen TTL von 7, da praktische Erfahrungen gezeigt haben, dass man damit 95% aller Knoten im Netzwerk erreichen kann. Unter der Annahme, dass es sich bei der Gnutella Struktur um einen Baum handelt, lässt sich nach [**Kab**] die maximale Anzahl der erreichbaren Hosts nach der folgenden Formel berechnen, wobei h die Höhe des Baumes (bzw. TTL) und d der Knotengrad (Anzahl der ausgehenden Kanten pro Knoten) ist:

$$f(d,h) = \sum_{t=1}^{h} (d-1)^{t-1} \cdot d$$

Bei einem angenommenen Knotengrad d = 8 und einem TTL = 7 sind nach dieser Formel theoretisch etwas mehr als eine Million Hosts von einem Knoten aus erreichbar. Dieser Formel liegt die Annahme zugrund das es sich bei Gnutella um einen strikten Baum handelt. In der Praxis enthält das Gnutella Netzwerk aber eine Menge Zyklen und skaliert deshalb bei weitem nicht so gut - zumeist waren in der Praxis nur etwa 40.000 Hosts zu erreichen.

Der eigentliche Prozess der Suche nach einer Datei und deren Download wird in Abb. 5 schematisch dargestellt.

Will man eine Datei im Gnutella-Netz finden, hat man aufgrund der fehlenden Struktur nur die Möglichkeit, eine „zufällige“ Suche zu starten, indem man eine Broadcast-Anfrage an alle Nachbarn verschickt. Diese vergleichen den Suchstring

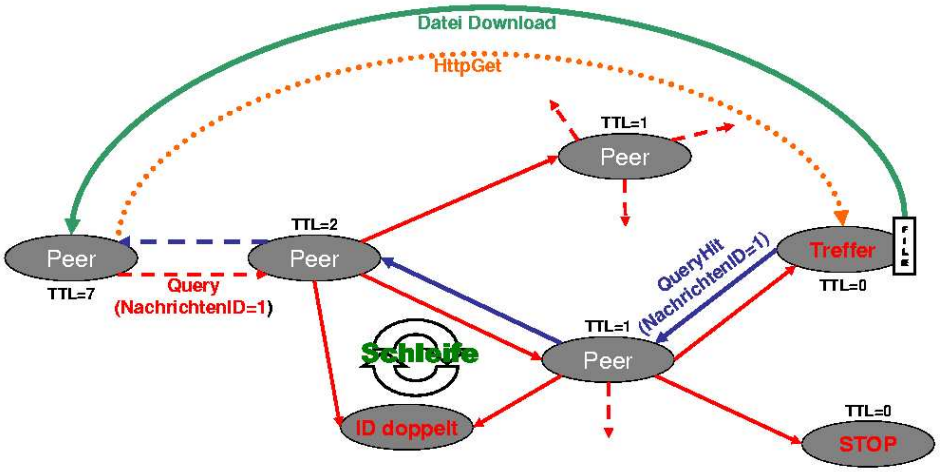


ABBILDUNG 5. Schematische Darstellung von Query, Reply und File Download bei Gnutella

mit den Namen ihrer lokalen Dateien. Wird eine Übereinstimmung festgestellt, wird eine QueryHit-Nachricht zum Absender der Query zurückgeleitet, wobei das gleiche Routingprinzip wie bei den zuvor erwähnten Ping-Nachrichten verwendet wird. Die Anfrage wird an alle direkten Nachbarn dieses Knotens weitergeleitet bis der TTL Null ist.

Nach dem Erhalt einer QueryHit-Nachricht stehen dem Knoten alle Information für das Starten eines direkten Out-of-Network Downloads zur Verfügung. Dies bedeutet, dass eine direkte Verbindung zwischen den beiden Hosts aufgebaut wird, wobei der Host, der die Nachricht enthält, als einfacher Web Server fungiert und über eine Standard Http-Get Nachricht die Datei dem suchenden Knoten liefert.

Da das „broadcast-and-forward“ Verkehrsprinzip auf den einzelnen Knoten zu einer starken Verkehrslast führt und durch die Verwendung eines TTL das Netz stark fragementier wird, skalierte das Gnutella Netzwerk natürlich nur schlecht. Aufgrund der zunehmenden Popularität von Gnutella und der damit verbundenen steigenden Zahl von Nutzern, kam es im August 2000 zum Zusammenbruch des Gnutella Systems. Dies lag daran, dass aufgrund der zunehmenden Anzahl der Benutzer der durch die Ping-/Pong-Signalisierung verursachte Kommunikationsoverhead so gross war, dass die Knoten keine anderen Aufgaben mehr erledigen konnten

Da die schlechte Skalierbarkeit in der dezentralisierten Architektur des Systems begründet war, wird bei aktuelleren Versionen von Gnutella (z.B. Morpheus) eine teilweise zentralisierte Struktur, wie ich sie im nächsten Abschnitt vorstellen werde, verwendet.

Teilweise zentralisierte unstrukturierte System. FastTrack, das Netzwerk auf dem die wohl populärste P2P-Applikation KaZaA aufbaut, ist als teilweise zentralisiertes, unstrukturiertes System aufgebaut. KaZaA verkündete Ende Mai [**Pat02**], das die Client-Software, die für das einloggen in KaZaA verwendet wird, alleine vom offiziellen Downloadserver 230 Mio. mal heruntergeladen wurde. Zählt man inoffizielle Server und die verschiedenen Klone der Software, wie KaZaA Lite oder Grokster, hinzu, ist KaZaA wohl die meistbenutzte Software aller Zeiten.

Teilweise zentralisierte Systeme basieren auf einem Zweischichten Konzept, da man neben den normalen Peers hier noch „Superknoten“ verwendet. Jeder Superknoten verwaltet einen Teilbereich des Netzwerks, indem er ein Verzeichnis aller Dateien der Peers in seinem Subnetz bereitstellt.

Dabei werden Peers vom System während des Logins dynamisch zu Superknoten ernannt wenn diese über genügend Bandbreite (z.B. DSL-Anbindung) und Rechenleistung verfügen. Wird man zum Superknoten ernannt, baut man Verbindungen zu anderen Superknoten auf und nimmt auch Anmeldungen gewöhnlicher Peers entgegen, wobei der Superknoten hierbei wie der zentrale Server in zentralisierten Systems eingesetzt wird. Wird man hingegen als normaler Peer eingestuft, sucht man sich einen Superknoten, der einem erlaubt eine Verbindung mit diesem einzugehen.

Hierdurch wird eine Zweischichten Architektur aufgebaut, in deren Zentrum sich die schnellen Superknoten befinden, die jeweils eine Menge von Peers verwalten. Diese Architektur ermöglicht das Routen von mehr Nachrichten, da der meiste Datenverkehr (Broadcast-and-Forward) über die schnellen Superknoten verläuft. Dadurch wird die Skalierbarkeit des Systems stark erhöht.

Stellt ein normaler Peer nun eine Suchanfrage, so wird dies zuerst zu seinem Superknoten weitergeleitet. Dieser Broadcastet die Anfrage zu all seinen Superknoten bis der TTL abgelaufen ist. Jeder dieser Superknoten hat ein Verzeichnis aller Dateien all seiner Peers und kann somit direkt über eine Übereinstimmung in seinem kompletten Teilbereich entscheiden. Da der TTL nur für Superknoten gilt und diese eine Menge Unterknoten enthalten, erreicht man mit KaZaA etwa 11-mal mehr Knoten als dies mit der ursprünglichen Version von Gnutella möglich war.

Insgesamt hat man mit dieser Architektur ein zuverlässigeres Backbone des Systems, was sich durch ein schnelleres Auffinden von Dateien bemerkbar macht. Außerdem weisen solche teilweise zentralisierten Systeme keinen zentralen Angriffspunkte auf, wie dies bei Napster der Fall war, da es eine Menge Superknoten gibt, welche dynamisch ihrer Aufgabe zugewiesen werden.

Lose strukturierte P2P-Systeme. Lose strukturierte Netzwerke verwenden Overlay Netzwerke zur Bestimmung der Position einer Datei im Netzwerk. Die in dieser Ausarbeitung beschriebenen Projekte verwenden dabei jedoch nicht den zuvor beschriebenen Ansatz der semantischen Overlay Netzwerke sondern einen Mechanismus der Dateien aufgrund von Hashfunktionen ihrer Inhalte im Netz verteilt. Diese Systeme zeichnen sich durch sehr gute Leistungen bei Punktanfragen (für die der exakte Schlüssel der Datei bekannt ist), während sie für Anfragen, die die Suchkriterien nur ungenau spezifizieren, wie z.B. Volltextsuche, nur bedingt geeignet sind.

Der wohl bekannteste Vertreter lose strukturierter P2P-Netzwerke ist Freenet, das 1997 von Ian Clarke, damals Student an der Universität von Edinburgh, als Abschlussarbeit initiiert wurde [**CSWH**]. Das Freenetprojekt (`http://freenetproject.org`) ist ein experimentelles Informationssharing-System ohne zentrale Kontrolle oder Administration und mit dem Hauptziel der völligen Anonymität für Herausgeber, Leser und Datentauscher.

Genau genommen handelt es sich bei Freenet nicht um ein File-Sharing System, sondern vielmehr um einen File-Storage Service, da Freenet eigentlich nur ungenutzten Speicherplatz auf den Knoten vereinigt und dadurch ein gemeinsames, verteiltes, virtuelles Dateisystem realisiert, das von unterschiedliche Anwendungen auf verschiedene Art und Weise genutzt werden kann. Für die Verwaltung dieses verteilten virtuellen Dateisystems enthält jeder Knoten im Freenet Netz eine lokale Datenbank und dynamische Routingtabellen, die von Applikation auf dem jeweiligen Knoten gelesen und beschrieben werden können.

Im Unterschied zu den unstrukturierten Netzwerken, die wir zuvor betrachtet haben, wird die Position der Dateien im Netzwerk bei Freenet durch einen binären Schlüssel festgelegt. Es gibt hierbei verschiedene auf dem Inhalt der Datei basierende Schlüsselverfahren, aber das bei Freenet am häufigsten eingesetzte Verfahren ist die Verschlüsselung des Dateinamens mittels eines SHA-1 (Secure Hash Algorithm)-Verfahrens, welches hier einen 160 Bit Hashwert aus dem Dateinamen generiert. Das Verfahren der Zuordnung einer Datei zu ihrem Knoten werde ich im nun folgenden Abschnitt erläutern.

Zuordnung einer Datei zu einem Knoten. Die Zuordnung einer Datei zu ihrem Knoten, welche letztlich die Overlay-Struktur des Freenet Netzwerkes realisiert, wird während des Speicherungsprozesses der Dateien durchgeführt. Dazu muss der Anwender zuerst mit Hilfe des SHA-1 Verfahrens den Schlüssel der Datei berechnen.

Ist dies geschehen, wird eine Insert-Nachricht zuerst an den eigenen Knoten gesendet. Die Insert-Nachricht enthält die zu speichernden Daten, den Dateischlüssel und einen TTL-Wert. Nimmt ein beliebiger Knoten auf dem Pfad vom Initiator bis zum Ablageort eine solche Insert-Nachricht entgegen, wird zuerst überprüft ob der Schlüssel bereits vergeben ist, d.h. ob lokal eine Datei mit gleichem Schlüssel existiert. Ist dies der Fall leitet der Knoten seine lokale Datei mit übereinstimmendem Schlüssel zum Initiator der Anfrage zurück und verhält sich so, als ob dieser eine Query nach der Datei mit diesem Schlüssel losgeschickt hätte. War der Schlüssel noch nicht vergeben, wird der Hashwert des Schlüssels mit den Hashwerten aller Dateien in der lokalen Routingtabelle des Knotens verglichen und aufgrund lexikalischer Dichte (z.B. minimaler Editierdistanz) der nächste Knoten auf dem Pfad ausgewählt. Ist das TTL-Limit ohne eine Schlüsselkollision erreicht, wird die Datei in diesem Knoten abgelegt und eine AllClear Nachricht zurückgeschickt, die dem Initiator mitteilt, dass der Speichervorgang erfolgreich war.

Während diesem Vorgang werden die lokalen Routingtabellen der Knoten stets auf den neuesten Stand gebracht, so dass der Rückweg anhand dieser Information wieder gefunden werden kann. Die Routingtabellen sind hierbei nachdem LRU

(Least Recently Used) Cacheverfahren organisiert, wodurch im Verdrängungsfall die am längsten nicht benutzen Einträge gelöscht werden.

Dieses strukturierte Ablegen der Dateien, basierend auf der lexikalischen Nähe des Dateischlüssels, führt dazu dass Dateien mit ähnlichen Schlüsseln auf denselben Knoten gespeichert werden.

Finden von Dateien. Der Suchvorgang bei Freenet unterscheidet sich durch den zusätzlichen Dateischlüssel stark von den zuvor vorgestellten Verfahren in unstrukturierten Netzwerken. Das Suchverfahren wird durch einen „steilstansteigenden Hill Climbing mit Backtracking“ Nachrichtenweiterleitungsmechanismus realisiert.

An Stelle eines Broadcast-Mechanismus, der eine Anfrage an alle umliegenden Nachbarn sendet, wie dies bei Gnutella der Fall ist, verwendet Freenet einen „chainmode“(Kettenmodus)-Mechanismus, um die Position der Datei zu bestimmen. Die Idee dahinter ist, dass die Anfrage nach einem Dateischlüssel von Knoten zu Knoten weitergeleitet wird, wobei jeder Knoten eine lokale Entscheidung über den nächsten Empfängerknoten der Nachricht trifft.

Erhält ein Knoten eine lokale Instanz der gesuchten Datei, d.h. der Schlüssel stimmt mit dem Schlüssel der Anfrage überein, ist die Suche beendet und die Datei wird mithilfe einer DataReply-Nachricht zum Initiator der Query auf dem gleichen Weg zurückgeleitet. Ein wichtiger Unterschied zu Gnutella ist hierbei, dass Freenet keine direkte Verbindung zwischen den am Datenaustausch beteiligten Peers aufbaut, sondern die Daten als expliziten Bestandteil der DataReply-Nachricht über alle Knoten des Pfades sendet. Dieser Verzicht auf eine direkte Verbindung zwischen den jeweiligen Knoten ist eines der Konzepte die die völlige Anonymität gewährleisten, da man nicht weiß ob eine Nachricht nun vom direkten Nachbar oder dessen Nachbarn kommt. Während dieses Dateitransports wird die Datei in jedem Knoten lokal gespeichert, so dass spätere Zugriffe schneller realisiert werden können. Auch die Datenbanken der Knoten sind hierbei nach dem LRU Prinzip aufgebaut, d.h. lange nicht benutzte Daten werden zuerst verdrängt. Dies führt dazu, dass unbenutzte Daten aus dem Freenet Netzwerk nach einer gewissen Zeit verschwinden während populäre Inhalte sich mehr und mehr verbreiten.

Ist die Datei allerdings nicht vorhanden, wird in der Routingtabelle des lokalen Knotens nachgeschaut und die dort vorhandenen Einträge mit dem aktuellen Suchkey verglichen. Aufgrund des Ablagemechanismus ist die Wahrscheinlichkeit am größten die gesuchte Datei auf dem Knoten zu finden, der in der Routingtabelle einen Schlüssel einer bereits gerouteten Datei enthält, deren lexikalische Dichte maximal ist. Die Query-Nachricht wird an diesen Knoten weitergeleitet. Um unendlich lange Ketten zu vermeiden, wird auch hier wiederum ein TTL verwendet.

Nun kann es natürlich vorkommen, dass eine Datei in einem aufgrund der maximalen lexikalischen Dichte gewählten Teilbaum nicht vorkommt, da es sich beim Routing ja nur um eine Schätzung der Position der Datei handelt. Hier kommt dann Backtracking zum Einsatz, d.h. man nimmt seine Entscheidung bis zu dem Knoten zurück der einem die Möglichkeit bietet einen anderen Weg zu beschreiten. Man untersucht also ausgehend von diesem Knoten den „nächstbesten“ Knoten. Dies geschieht solange, bis man die Datei entweder gefunden hat oder aber im schlimmsten Falle bis man alle Knoten abgesucht hat, was bedeutet das eine Datei mit diesem Key innerhalb des durch den TTL erreichbaren Bereichs nicht vorhanden ist. Die Anzahl der Knoten, die in diesem Falle untersucht werden, stimmt mit der Anzahl der durch Broadcast untersuchten Knoten in unstrukturierten Netzwerken überein.

In der folgenden Abbildung 6 wird der Ablauf eines Suchvorganges nochmals schematisch dargestellt. Im Schritt 1 wurde im Knoten 1 lokal die Entscheidung gefällt, dass die Datei sich in Richtung Knoten 2 befindet. Da dies eine Fehlannahme war und die Datei sich in keinem der Subknoten von Knoten 2 befand, wird

Backtracking eingesetzt (gekennzeichnet mit (1)). Nun wird geschätzt, dass sich die Datei mit größter Wahrscheinlichkeit in Richtung von Knoten 4 befindet (2). Diese Schätzungen setzen sich fort bis man mit (4) den passenden Knoten findet. Die Datei wird nun zum Initiator der Query über denselben Weg zurücktransportiert und dabei in jedem Knoten unterwegs lokal abgelegt (Caching), d.h. dass nach diesem Vorgang jeder Pfadknoten (1, 4, 5, 7, 9) eine lokale Instanz der Datei enthält. Knoten 3, 6 und 8 wurden in dieser Abfrage nie beschritten, da die Dichte ihrer Schlüssel nicht ausreichend war (3).

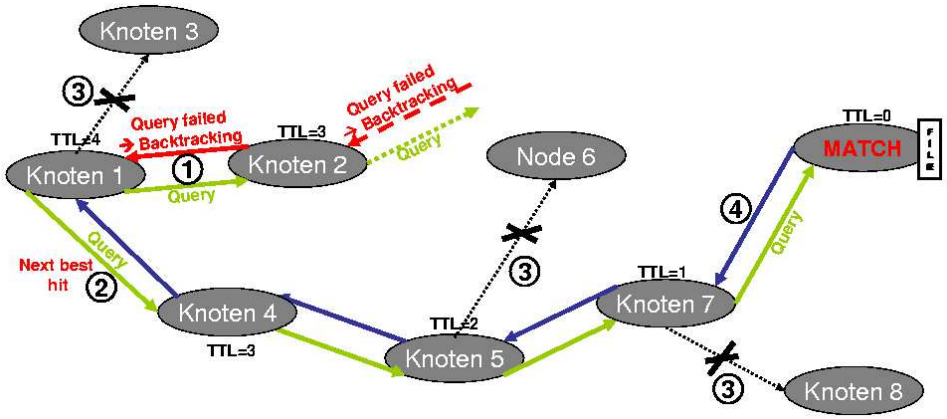


Abbildung 6. Der Freenet Chain-Mode Mechanismus zur Suche von Dateien

Das Freenet Netzwerk wird als „lose strukturiert“ klassifiziert, da ausgehend von den Informationen die ein Knoten in seinem Stack hat, er nur die ungefähre Position der Datei schätzen kann. Dadurch vermeidet man zwar das blinde Broadcasting der unstrukturierten Systeme, aber einen eindeutigen Weg von Quelle zu Ziel ohne Backtracking wird man nur in den wenigsten Fällen finden.

Das größte Problem von Freenet ist jedoch das Auffinden von Dateien, da man hierzu den exakten Schlüssel, wie z.B. den Dateinamen, exakt kennen muss. Bereits kleinste Unterschiede in diesem Schlüssel ergeben einen anderen Hashwert und folglich kann eine vorhandene Datei, die netzintern nur durch diesen Schlüssel identifiziert wird, nicht gefunden werden. Bis heute wird zur Bekanntgabe neuer Dateien und deren Schlüssel meist ein so genanntes Offlineverfahren, wie z.B. Mailinglisten oder webbasierte Kataloge, verwendet.

Strukturierte Systeme. Wie bereits zuvor erwähnt, zeichnen sich strukturierte Netzwerken durch eine strenge Kontrolle auf der Ebene des Overlay Netzwerks aus. Diese Kontrolle sorgt dafür, dass Dateien oder Zeiger auf diese nur an genau festgelegten Positionen im Netzwerk abgelegt werden können.

Pastry [**RDb**]. Pastry ist laut Definition eine „skalierbare selbstorganisierende P2P Routing- und Objektlokalisierungs-Infrastruktur für P2P-Anwendungen". Pastry ist somit keine direkte P2P Anwendung sondern dient als Grundlage für die Realisierung des Overlay Netzwerkes strukturierter P2P-Applikationen.

Jeder Pastry Knoten wird eindeutig identifiziert durch eine 128-bit Knoten-ID. Die Knoten-ID wird verwendet um die Position eines Knoten in einem zirkulären Knoten-ID-Raum anzugeben. Die Menge der Knoten-IDs sollte hierbei gleichmäßig im aufgespannten Schlüsselraum verteilt sein. Die Vergabe der Knoten-IDs ist Aufgabe der Applikation, die auf Pastry aufsetzen. Bei der ID könnte es sich z.B. um den kryptographischen Hashwert des öffentlichen Schlüssels des Knotens oder um dessen IP-Adresse handeln.

Auch die Nachrichten werden durch ein Verfahren wie dies bereits bei Freenet vorgestellt wurde, durch eine 128 Bit Nachrichten-ID eindeutig gekennzeichnet.

Da Pastry Netzwerklokalität zur Verminderung der Netzlast ausnutzt, muss gewährleistet sein, dass je zwei Knoten ihren Abstand zueinander messen können. Dies könnte z.B. durch das Zählen der IP Routinghops die ein Paket zwischen 2 Pastry Knoten zurücklegen muss, realisiert werden, da sich Nachbarn im Pastry Overlay in der Anzahl der zwischen ihnen befindlichen physikalischen Hops (z.B. IP) unterscheiden können.

Jeder Pastry Knoten verwaltet drei Zustandsmengen: eine „**Blattmenge**", eine „**Nachbarschaftsmenge**" und eine **Routingtabelle**. Die Blattmenge enthält L Knoten, wobei es sich bei diesen Knoten um die numerisch dichtesten Knoten (bezogen auf die Knoten-ID) handelt, wobei L/2 hiervon die numerisch dichtesten kleineren Knoten-IDs und $\frac{L}{2}$ die numerisch dichtesten größeren Knoten-IDs sind. Die Nachbarschaftsmenge enthält die am wenigsten weit entfernten Knoten bezogen auf die Netzwerkdistanz. Die Routingtabelle ist nach dem Prinzip des Präfixrouting realisiert und enthält für jede Stelle des ID-Raumes eine eigene Zeile. Die Zeile n beinhaltet eine Liste von Knoten, die in den ersten n Stellen mit dem Präfix des aktuellen Knotens übereinstimmt, sich aber an der Stelle n+1 von diesem unterscheidet. Gibt es mehrere Knoten die diese Vorgabe erfüllen wird der am wenigsten weit entfernte Knoten im Netz eingetragen.

Soll nun eine Nachricht, die durch eine entsprechende Nachrichten-ID gegeben ist, durch das Netz geroutet , überprüft der Knoten der die Nachricht entgegennimmt zuerst, ob die Nachrichten-ID in seiner Blattmenge enthalten ist. Ist dies der Fall wird die Nachricht direkt zum entsprechenden Zielknoten weitergeleitet. Sonst wird die Nachricht in jedem Schritt an den Knoten weitergeleitet, dessen ID numerisch am dichtesten zur gegebenen Nachrichten-Key liegt. Hierzu wird das Prinzip des Präfix-Routing verwendet, d.h. eine Nachricht wird stets an einen Knoten weitergeleitet, dessen Übereinstimmung zwischen dem Präfix der Knoten-ID und der Nachrichten-ID um mindestens eine Stelle länger ist als dies im aktuellen Knoten der Fall ist. Gibt es einen solchen Knoten nicht, wird die Nachricht zu einem Knoten weitergeleitet, dessen Präfix mit der Nachrichten-ID in genauso vielen Stellen übereinstimmt wie der Präfix des aktuellen Knotens, dessen numerische Dichte jedoch größer ist.

Nimmt man nun an, dass ein Netzwerk aus N Knoten besteht, kann Pastry im normalen Betrieb eine Nachricht mit entsprechender ID in $\lfloor log_{2^b} N + 1 \rfloor$ Schritten zum numerisch dichtesten Knoten routen, wobei b ein Konfigurationsparameter ist dessen typischer Wert 4 beträgt.

Trotz etwaiger Knotenausfälle, ist die Auslieferung der Nachricht garantiert wenn nicht $\lfloor \frac{|L|}{2} \rfloor$ benachbarte Knoten (mit ähnlichen KnotenIDs) gleichzeitig ausfallen. L ist dabei die Mächtigkeit der Blattmenge des entsprechenden Knotens. Da bei diesem Verfahren stets auch die Distanz im Netzwerk minimiert wird und Suchanfragen (bei geg. Schlüssel) sehr effizient durchgeführt werden können, skaliert dieses System, verglichen mit unstrukturierten Netzwerken, außerordentlich gut.

Verschieden Applikationen verwenden Pastry als Routinginfrastruktur. Im nun folgenden Abschnitt werde ich nur ganz kurz PAST, ein P2P File Sharing System basierend auf Pastry, skizzieren. Außerdem dient Pastry auch als Objektlokalisierungsinfrastruktur für Pastiche, das in Abschnitt 4 genauer betrachtet wird.

PAST [**RDa**]. PAST ist ein P2P-System zur verteilten Speicherung von Dateien das Pastry als Infrastruktur zu Lokalisierung von Objekten verwendet. Hierbei verwendet PAST eine Datei-ID, die sich aus einem Hashwert des Dateinamens und dem Eigentümer der Datei zusammensetzt, zur Bestimmung der Position einer Datei im Netz. Die Kopie einer Datei wird hierbei auf den am wenigsten weit entfernten k Pastryknoten gespeichert, die eine max. numerische Übereinstimmung zwischen Knoten-ID und DateiID besitzen. Somit kann eine Datei gefunden werden, indem man den Dateischlüssel als (ungefähren) Schlüssel des entsprechenden Knotens betrachtet. Wenn nicht alle k Knoten, die die Datei lokal enthalten, gleichzeitig ausfallen, wird durch den Einsatz von Pastry erreicht, dass eine vorhandene Datei im Schnitt in $\lfloor log_{2^b} N + 1 \rfloor$ Schritten gefunden wird. Hierbei handelt es sich dann um einen Knoten mit minimaler Distanz zum Anfrageknoten, was zu einer Reduzierung der Netzlast führt.

Doch warum wird dieses System als strukturiert und nicht wie Freenet, das auf einem ähnlichen Routingprinzip basiert, als lose strukturiert klassifiziert? Freenet verwendet für die Bestimmung des Ablageorts seiner Dateien ein Verfahren, dass aufgrund der lexigraphischen Dichte von Dateischlüsseln Dateien mit ähnlichen Schlüsseln auf Knoten gruppiert. Auch in PAST werden Knoten mit ähnlicher ID auf denselben Knoten abgelegt, doch darüber hinaus liegt auch die KnotenID numerisch dicht zu den IDs der Dateien, die in diesem Knoten abgelegt werden. Somit findet bei der Suche nicht nur ein Schätzen des Pfades aufgrund der Dateien in den Knoten statt, sondern man vergleicht direkt die Datei-ID mit der Knoten-ID der Nachbarknoten um eine Query-Nachricht in die richtige Richtung zu routen. Darüber hinaus verwendet Pastry auch kein TTL, so dass eine genauere Übereinstimmung zwischen Datei-ID und Knoten-ID erreicht wird als dies bei Freenet der Fall ist. Alles in allem erreicht man eine starke Strukturierung der Daten im Netz.

Für genauere Details dieser Technologie sie auf [**RDa**] verwiesen

3.3. Distributed Processing (Verteiltes Rechnen). Systeme, die auf dem Paradigma des verteilten Rechnens basieren, machen sich ungenutzte Rechenleistung auf Desktop PCs zunutze, um komplexe Berechnungen durchzuführen, die ansonsten nur auf sehr kostenaufwändigen Supercomputern möglich sind. Dazu wird auf dem Desktop PC eine Applikation installiert, die sich als eine Art spezieller Bildschirmschoner präsentiert und die dafür sorgt, dass die Rechenleistung des PCs in Ruhezeiten der Berechnung einer größeren, verteilten Aufgabe zugeführt wird.

Solche Systeme werden meist eingesetzt, um umfassende Berechnungen öffentlicher Forschungsprojekte durchzuführen. Sie können aber auch durchaus für firmeninterne Berechnungen verwendet werden. Beispiele für den Einsatz solcher Systeme sind z.B. das „Compute Against Cancer"-Projekt, das dem Vergleich von Proteinsequenzen für die Krebsbekämpfung dient. Der wohl bekannteste Vertreter ist aber SETI@HOME, das für die Suche außerirdischer Intelligenz eingesetzt wird und auf den ich in den folgenden Zeilen kurz eingehen werde.

Fallbeispiel: SETI@HOME. SETI@HOME ist Bestandteil eines Projekts zur Suche extraterrestrischer Intelligenz und wird für die Analyse von Datensignalen, die von dem 305 Meter großen Satellitenteleskop Arecibo in Puerto Rico aufgenommen werden, verwendet. Hierbei werden die aufgenommenen Signale auf ungewöhnliche Radiowellen untersucht, die ein Zeichen für außerirdische Kommunikation darstellen könnten.

SETI@HOME vereinigt hierzu die Rechenleistung von etwa 500.000 Peers, was zu einer verteilten Rechenleistung von etwa 25 Teraflops führt. Somit gehört dieses System zu den schnellsten Supercomputern der Welt. Aufgrund des Einsatzes kostenloser Rechenleistung auf normalen Desktop PCs sind die Betriebskosten von SETI@HOME äußerst gering. Das initialen Ausgaben beliefen sich auf etwa eine halbe Million Dollar. Zum Vergleich: Für ANSI White, einer der leistungsfähigsten Supercomputer der Welt mit seiner Leistung von 12 Teraflops liegt der Anschaffungspreis bei etwa 110 Mio. $ (Stand Anfang 2002).

Architektonisch betrachtet handelt es sich bei SETI@HOME um eine Client-Server-Architektur. Der zentrale Server speichert hierbei alle Daten die vom Teleskop Arecibo gesammelt werden und zerlegt diese Datensignale in kleinere Pakete, die dann von den Klienten heruntergeladen werden. Die Klienten, d.h. die Desktop Computer, untersuchen dann in der Zeit, in der sie nicht genutzt werden, diese Datenpakete und laden diese nach abgeschlossener Berechnung wieder auf den Server zurück. Der Server hat dann nur noch die Aufgabe die Ergebnisse der Berechnung zu überprüfen.

Nun kann man sich natürlich fragen, was an diesem Konzept denn P2P sein soll, da es keinerlei direkte Verbindung zwischen zwei ebenbürtigen Peers gibt. Vielmehr macht es den Eindruck, als ob es sich hierbei um eine übliche Client/Server-Architektur handelt, in der die Klienten die Daten herunterladen, diese ändern und wieder zurück auf den Server legen. Doch das stimmt nicht ganz. Im Gegensatz zum üblichen Client-Server-Paradigma sind bei SETI@HOME die Klienten weit mehr als nur dumme Browser. Sie stellen einen zentrale, aktive Rolle in der Funktionsweise des Systems dar, da sie einen kleinen Teil einer großen Aufgabe autonom (Kontaktaufnahme zum Server nur bei Down- bzw. Upload der Pakete) berechnen und der zentrale Server nur für administrative Aufgaben zwischen den gleichberechtigten Klienten verwendet wird. Ohne die Klienten hätte das System keinerlei Funktionalität.

3.4. Instant Messaging (IM). IM hat sich zum Ziel gesetzt, direkte Kommunikation zwischen Personen zu ermöglichen. Eine Zielvorgabe war, dass IM-Systeme dem Anwender durch Realisierung direkter Person-zu-Person Kommunikation (über direkte P2P Komminukationskanäle) ein Gefühl der menschlichen Nähe vermitteln sollen.

Dieses Konzept der menschlichen Nähe wird unterstützt durch den Aufbau von „Freundesgruppen“, denen sich ein Benutzer anschließen kann. Nimmt ein Anwender an einem IM-Prozess teil, kann er seine Anwesenheit signalisieren und jedem der anderen Teilnehmer wird die entsprechende Person und deren Status angezeigt.

Neben diesen Freundesgruppen ermöglichen IM-Systeme auch direkte Kommunikationen zwischen zwei beliebigen Anwendern. Kommunikation beschränkt sich hierbei meist auf textuelle Verfahren, der sogenannte „Chat“. Es gibt aber auch Systeme die Audio- und Videoübertragungen zwischen Anwendern ermöglichen.

Die wohl bekanntesten Peer-to-Peer IM-Systeme sind AOLś Instant Messenger, Yahoo Messenger!, Microsoft MSN Instant Messenger und ICQ. Jede dieser zuvor genannten Applikationen basiert auf P2P-Verfahren.

3.5. Collaboration (Zusammenarbeit). Bei Collaboration-Systemen handelt es sich ganz allgemein um P2P-Systeme, die einen verteilten Arbeitsbereich gemeinsamen nutzen. Diese P2P-Systeme haben eine Menge gemeinsam mit gängigen Groupware-Systemen, die räumlich verteilten Anwendern die Zusammenarbeit an gemeinsamen Projekten unabhängig von räumlicher und/oder zeitlicher Trennung ermöglichen.

Ein Beispiel für ein solches P2P-Collaboration Projekt werde ich im folgenden Abschnitt anhand eines aktuellen Forschungsprojektes der Universität von Michigan - genannt Pastiche - behandeln. Pastiche stellt ein einfaches und kostengünstiges Verfahren zur Sicherung von Daten (Backup), durch die optimierte Nutzung ungenutzten Speicherplatzes auf Clientsystemen, zur Verfügung.

3.6. Zukunft von P2P. Um die Zukunft von P2P bewerten zu können, muss man zwischen Anwendungen für den einfachen Endbenutzer und dem Einsatz von P2P zur Unterstützung von Businessszenarien großer Firmen unterscheiden.

Auf dem Endanwendermarkt ist P2P zweifellos auf dem Vormarsch. Heutige P2P-Anwendungen leiden dabei jedoch meist unter größeren konzeptionellen Problemen, die eine weitere Verbreitung erschweren, wie z.B. die mangelnde Skalierbarkeit und Robustheit. Außerdem ist das Vertrauen in die Integrität solcher Systeme meist nicht gegeben, da man z.B. nie weiß ob die angeforderte Datei wirklich das enthält, was sie vorgibt zu enthalten. Bei Gruppenkommunikation oder ähnlichen Anwendungen zwischen Usern fehlen auch zumeist entsprechende Authentifizierungsmechanismen, welche die Identität der Benutzer überprüft.

Darüber hinaus existiert derzeit noch kein Bezahlmodell in P2P-Systemen, so dass die Entwicklung neuer Massenmarkt-Anwendungen für Firmen wenig lukrativ ist.

Werden diese Probleme gelöst, steht einem weiteren Durchbruch von P2P-Systemen für den einfachen Benutzer in Form unterschiedlichster Anwendungen nichts mehr im Wege.

Um der Verwendung von P2P-Systemen auch im Geschäftsbereich zum Durchbruch zu verhelfen, sind wohl etliche Anpassungen an gegenwärtigen P2P-Systemen notwendig. Grundsätzlich widerspricht der Grundgedanke von P2P-Systemen, der einen dezentralisierten - und somit auch nicht zentral kontrollierbaren - Austausch von Informationen zwischen gleichberechtigten Partnern vorsieht, dem Konzept des Schutzes von gutbehüteten Firmengeheimnissen. Somit steht P2P vor den gleichen Hürden wie ehemals die Open Source-Bewegung. Eine Menge großer Firmen, wie Intel, Compaq, HP, Sun und IBM arbeiten derzeit an P2P-Projekte, mit dem Ziel diesen Technologie zum kommerziellen Durchbruch zu verhelfen. Die größte Herausforderung hierbei dürfte sein, ein mächtiges Geschäftsmodell zu entwickeln welche die Vorteile von P2P auszuschöpfen weiß.

Die Vermutung liegt nahe, dass gegenwärtige P2P-Entwicklungen nicht zu einem großen, universellen Netzwerk, in dem jeder Anwender mit anderen Anwendern kommunizieren kann, sondern zu einer heterogenen Ansammlung nicht kompatibler P2P-Netzwerke mit unterschiedlicher Verwendungszwecken führen wird.

Diese heterogenen Netzwerke, die nur einer gewissen Klientel bereitstehen werden und keine zentrale Kontrolle oder Organisation aufweisen, müssen Möglichkeiten bieten zur Durchsetzung netzinterner Regeln. Dies beinhaltet zum Beispiel Konzepte zur Benutzerauthentifizierung, benutzerabhängige Definition von Rechten bzw. Regeln und Ausschluss von Personen, die diese Regeln verletzten. Ohne zentrale Kontrollinstanz ist dies die gegenwärtig größte Herausforderung vor der die Entwickler neuer Generationen von Peer-to-Peer Systemen stehen.

4. Fallbeispiel für Collaboration: Pastiche

Pastiche[**CMN**] ist ein Projekt an der Universität von Michigan, mit dem Ziel ein einfaches, billiges und auf P2P basierendes System für die Datensicherung (Backup) zur Verfügung zu stellen. Backup in seiner heutigen Form ist ein sehr umständlicher und teurer Prozess und stellen für Firmen einen großen Ausgabenposten dar. Ein Kostenbeispiel zur Verdeutlichung: Bei Connected TLM, einer Firma die sich auf den Schutz von Daten spezialisiert hat, kostet ein reiner Daten-Backupdienst mit einem max. Datenvolumen von bis zu 4 GB etwa 15$ pro Monat.

Der erhebliche Kosten- und Arbeitsaufwand führen oft dazu, dass im privaten Bereich meist überhaupt keine Backups gemacht werden und auch Firmen Backups auf das Nötigste (z.B. Kundendatensätze) beschränken, was evtl. gravierende Folgen mit sich bringen kann.

Der Hauptvorteil von Pastiche gegenüber herkömmlichen Backup-Systemen ist, dass der benötigte Speicherplatz für die Backupdaten durch freien Festplattenspeicherplatz von Endsystemen (Desktop PCs) zur Verfügung gestellt wird. Aktuelle Untersuchungen von etwa 5.000 Computern haben ergeben, dass nur etwa 53% der Speicherkapazität genutzt wird. Durch die Verwendung dieses „überschüssigen" Speicherplatzes, fallen nur minimale Kosten für den Backup-Speicherplatz an und auch der Administrationsaufwand soll auf ein Minimum beschränkt werden.

Die Knoten im Pastiche Netzwerk bilden also eine kooperierende, wenn auch unzuverlässige Netz von Knoten, die sich gegenseitig für das Sicherung ihrer Daten zur Verfügung stehen. Da die einzelnen Maschinen sich im Netzwerk an- und abmelden können wie es ihnen beliebt, müssen die Daten jedes Pastiche Knotens auf mehr als nur einem zusätzlichen System archiviert sein. Typischerweise werden die Daten auf 5 unterschiedlichen Systemen abgelegt. Die meisten dieser Kopien werden hierbei auf nicht weit entfernten Systemen (bzgl. der Netzwerkdistanz) abgelegt, um die Netzwerklast zu minimieren. Es ist vorgesehen ein Kopie auf einem weiter entfernt liegenden Rechner zu archivieren, um gegen eine Katastrophe geschützt zu sein.

Zur Optimierung des Backup-Prozesses sieht Pastiche einen Mechanismus zur Minimierung des Speicheraufwandes vor. Dieser besteht darin, dass man für jeden Knoten entsprechende Backup-Peers sucht, die bereits eine max. Übereinstimmung mit den Daten des aktuellen Knotens haben. Dieser Mechanismus beruht auf Untersuchungen die zeigen, dass viele System sehr ähnliche Datenbestände enthalten und dass die meisten dieser Daten bereits zur Installationszeit erzeugt werden. Die Standardinstallation von Office 2000 benötigt z.B. ca. 217 MB und ist beinahe allgegenwärtig. Durch die Ausnutzung solcher gemeinsamer Datenbestände auf unterschiedlichen Systemen lassen sich natürlich erhebliche Optimierungen erzielen.

4.1. Verwendete Technologien.

Pastiche basiert auf drei relativ neuen Technologien:

- **Pastry**, eine P2P Routing und Objektlokalisierungs-Infrastruktur für P2P-Anwendungen (bereits vorgestellt)
- **Kontextbasierte Indizierung** zur Erkennung redundanter Daten in Dateien
- **Konvergente Verschlüsselung** zur Benutzung der gleichen verschlüsselten Darstellung eines Datums ohne notwendigen Schlüsselaustausch.

Kontextbasierten Indizierung wird von Pastiche eingesetzt, um den Speicheroverhead zu minimieren, in dem redundante Datenblöcke in unterschiedlichen Dateien innerhalb des eigenen Systems oder in Datenbeständen unterschiedlicher Peers gefunden werden. Die Herausforderung hierbei ist das Finden von Überschneidungen in augenscheinlich unabhängigen Dateien, ohne die zugrunde liegende Struktur dieser Dateien zu kennen. Kontextbasierte Indizierung realisiert dies durch die Identifikation von Grenzregionen, so genannten Ankern, unter Verwendung von Rabin

Fingerprints. Diese Fingerprints werden für jeden überlappenden k-Byte Substring einer Datei bestimmt. Stimmen die niederwertigen Bits dieses Fingerprints mit einem vordefinierten Wert überein, wird dieser Offset als Anker markiert. Die Anker unterteilen eine Datei in *Chunks* (dt. Brocken).

Da der eigentliche Backupprozess auf Grundlage dieser Chunks geschieht, wird bei Pastiche ein neues Dateisystem namens *Chunkstore* eingesetzt. Chunkstore speichert allen Daten - sowohl die eigenen als auch die Backupdaten anderer Systeme - in Einheiten die für das Sharing ideal sind, ohne die Leistung des eigentlichen Systems merklich zu beeinträchtigen. Zur Identifizierung der jeweiligen Chunks wird ein inhaltsbasiertes Hashverfahren verwendet. Somit haben alle Chunks mit gleichem Inhalt die gleiche ID, unabhängig von ihrem Speicherort.

Durch das Backup der Daten auf Systemen anderer Anwender, muss Pastiche für die Vertraulichkeit und Integrität der Daten aller Teilnehmer sorgen. Zur Verschlüsselung der Daten müssen alle Teilnehmer ein einheitliches Schlüsselprotokoll verwenden. Würde nämlich jeder Benutzer seinen eigenen krytographischen Verfahren verwenden, würden Chunks mit gleichem Inhalt unterschiedlich dargestellt werden, was das Sharing dieses Chunks verhindern würde.

Für diesen Zweck sieht Pastiche den Einsatz konvergenter Verschlüsselung vor. Eine schematische Darstellung des Ablaufs kann der folgenden Abbildung 7 entnommen werden. Hierbei wird jeder Chunk c mittels SHA-1 gehashed. Das Resultat dieser Hashwertberechnung H_c, wird als Chunkś Handle bezeichnet. Dieser Handle wird nun verwendet um einen symmetrischen (privaten) Schlüssel K_c, mittels eines geheimen Schlüsselgenerierungsverfahrns zu erzeugen. Die Anwendung des privaten Schlüssels auf Klartextchunk ergibt eine verschlüsselte Darstellung des Chunks, die nur vom Besitzer des Chunks selbst gelesen werden kann. Damit Chunks mit identischem Inhalt die selbe Repräsentation haben, wird der Handle H_c erneut gehashed, was die öffentliche Chunk-ID I_c, des Chunks ergibt. Jeder Chunk wird somit verschlüsselt mit dem geheimen K_c unter der ID I_c gespeichert.

Wird ein entsprechender Chunk nun zwischen verschiedenen Usern geshared, so gibt es für jeden Anwender einen geheimen Dateischlüssel K_c, aber nur eine Repräsentation des verschlüsselten Chunks, die ID I_c. Die Entschlüsselung des Chunks erfolgt unter Verwendung des privaten Schlüssels und anschließende Anwendung des SHA-1 Verfahrens.

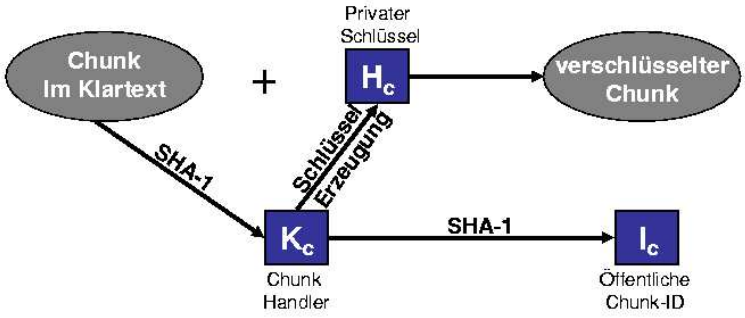


Abbildung 7. Schema der konvergenten Verschlüsselung

Trotz dieser konvergenten Verschlüsselung gibt es in Pastiche eine Lücke in der Vertraulichkeit. Sichert ein Knoten i einen Chunk auf einem anderen Knoten j, der diesen Chunk nicht lokal vorliegen hat, so kann j die Datei nicht lesen da er den privaten Schlüssel von i nicht kennt. Liegt in j die Datei aber lokal vor, so kennt j den Inhalt dieses Chunks, da es ja auch Bestandteil seiner Daten ist, und er weiß, dass i diese Daten auch besitzt. Diese Lücke wurde jedoch aus Performanzgründen

in Kauf genommen.

4.2. Design von Pastiche. Wie bereits zuvor erwähnt, werden Daten in Pastiche als Chunks gespeichert, wobei deren Grenzen durch kontextbasierte Indizierung festgelegt werden und die Verschlüsselung mittels konvergenter Verschlüsselung erfolgt. Jeder Chunk umfasst eine Eigentümer-Liste, die die ID aller Knoten enthält die Referenzen auf den entsprechenden Chunk haben. In der folgenden Beschreibung gibt es keinen Unterschied zwischen dem Speichervorgang eines Chunks auf einem lokalen und/oder auf einem Backup-System.

Wird eine neue Datei gespeichert, wird diese zuerst in Chunks zerlegt. Bevor der Chunk nun auf die Platte geschrieben werden soll, überprüft Pastiche durch Vergleich der öffentlichen Chunk-ID I_c mit allen Chunk-IDs des Systems ob dieser Chunk dort nicht bereits vorhanden ist. Wenn der Chunk bereits vorhanden ist, wird der Verfasser der Datei zu der Eigentümerliste hinzugefügt und ein lokaler Referenzzähler wird erhöht. Andernfalls wird der Chunk mit dem privaten Schlüssel K_c verschlüsselt und mit einem Referenzzählerwert von 1 (für den lokalen Benutzer) auf die Platte geschrieben. Hierbei wird das Zerlegen in Chunks und das Schreiben auf die Festplatte grundsätzlich verzögert durchgeführt, um Overhead durch Chunkberechnungen für kurzlebige/temporäre Dateien zu vermeiden. Natürlich nimmt man für diesen Performanzgewinn eine schwächere Persistenz in Kauf.

Die Liste von Chunk-IDs, die das aktuelle Dateisystem eines Knotens beschreiben, wird als Signatur bezeichnet.

Daten Chunks sind grundsätzlich unveränderlich, d.h. eine Änderung eines Chunks durch editieren des entsprechenden Inhalts eines Files führt zur Erzeugung eines neuen Chunks und der Erniedrigung des Referenzzählers. War dies die einzige Referenz auf diesen Chunk für den entsprechenden Anwender, wird dieser aus der Eigentümerliste entfernt. Entsprach der Benutzer dem letzten Eintrag in der Eigentümerliste, kann der Chunk gelöscht werden.

Jede Datei, die sich ja aus einer Vielzahl von Chunks zusammensetzt, wird durch Metadaten beschrieben, die neben den üblichen Eigenschaften wie Eigentümer, Rechte, Erstellungs- und Modifizierungsdaten, eine Liste von Chunk Handels H_c enthalten, die die in der Datei beinhalteten Chunks identifiziert. Da diese Chunk Handels genutzt werden, um den privaten und den öffentlichen Schlüssel zu generieren, müssen die Metadaten zur Gewährleistung der Sicherheit, verschlüsselt werden. Diese Metadaten werden von Pastiche nicht in Chunks zerlegt, da diese nur geringe Datenmengen umfassen und nur selten geshared werden können. Auch sind diese Metadaten veränderbar, da ansonsten jede kleine Änderung des Inhalts eine neue Instanz der Metadaten eines Files erzeugen müsste, was dann wiederum zu einer Änderung im übergeordneten Verzeichnis führen würde und so weiter. Somit wird der Handel H_c und die Schlüssel I_c und K_c für die Metadaten einer Datei, nur zum Zeitpunkt des Anlegens der Datei erzeugt und danach stets wiederverwendet.

Um ein komplettes Backup eines Rechners zu realisieren, werden die Metadaten, die die Wurzel des Systems beschreiben, gesondert behandelt. Der Chunkhandel H_c dieser Metadaten wird mithilfe eines Anwenderpasswortes verschlüsselt. Wie man später sehen wird, wird für die komplette Wiederherstellung des Systems somit dieses Passwort und der Rechnername benötigt.

Um böswilliges Löschen von Backupdaten durch Dritte zu verhindern, integriert Pastiche ein typisch Signaturverfahren der asymmetrischen Verschlüsselung, bei dem eine entsprechende Löschanforderung mit dem entsprechenden privaten Schlüssel des Senders unterzeichnet werden muss. Durch Entschlüsselung der Signatur mit dem öffentlichen Schlüssel des Senders kann die Authentizität des Absenders überprüft werden.

Finden von „Buddies“ unter Verwendung der Pastry Overlays. Das Finden geeigneter Buddies, wie die Knoten die das Backup realisieren bezeichnet werden, ist einer der zentralen Punkte von Pastiche.

Buddies sollten zwei Eigenschaften erfüllen:

- die lokalen Daten der Buddies sollten eine max. Übereinstimmung mit den Daten auf dem entsprechenden Knoten auweisen, um den Speicheraufwand zu minimieren
- minimale Netzwerkdistanz zur Reduzierung der Netzwerklast.

Für die Suche nach geeigneten Buddies werden zwei unterschiedliche Pastry-Overlaynetzwerke eingesetzt. Beim ersten handelt es sich um ein Standard Pastry-Netzwerk, dessen Organisationsstruktur auf minimaler Distanz der Knoten im Netz beruht, während das zweite Overlay auf dem Überdeckungsgrad der Daten auf den Knoten basiert.

Jeder Knoten des Pastiche Netzwerkes wird standardmäßig in das Pastrynetzwerk, das nach dem Prinzip der Netzwerkdistanz organisiert ist, eingebunden. Die Knoten-ID ist hierbei der vollständige Domainname des Rechners.

Nachdem der Knoten eingebunden wurde, wählt er eine beliebige (fiktive) Knoten-ID und sendet eine Discovery Anfrage an diesen Knoten. Die Discovery Anfrage enthält eine Menge spezieller Chunk-IDs, welche die Datenstruktur des Senderknotens beschreiben und mit deren Hilfe ein potentieller Buddy seinen Überdeckungsgrad bestimmen kann. Jeder der Knoten, der auf dem Weg zu dem eigentlichen Zielknoten beschritten wird, berechnet seinen Überdeckungsgrad und sendet eine Antwort zurück. Ergibt die erste Anfrage keine zufriedenstellende Menge von Buddies wird die Anfrage wiederholt, indem die erste Stelle des ersten gewählten Zielknotens für das Discovery Paket verändert wird. Da Pastry Präfixrouting verwendet wird hierdurch ein anderer Pfad im Netzwerk beschritten. Dieses Verfahren wird solange wiederholt bis entweder eine Menge von Buddies mit ausreichender Datenübereinstimmung gefunden wurde oder aber alle Knoten des ersten Pastry-Netzwerks (mit minimaler Distanz) überprüft wurden.

Knoten, die über eine allgemein übliche Installation (Betriebssystem, Software) verfügen, sollten keinerlei Schwierigkeiten haben, geeignete Buddies zu finden. Für Knoten, die über ein ausgefallenes Betriebssystem und damit verbundene Software verfügen, kann dies jedoch schon schwieriger sein. Für solche Knoten gibt es aber das zweite Pastry-Overlay, welches den Überdeckungsgrad der Daten auf den Knoten statt der Netzwerk-Hops als Distanzmetrik verwendet. Der neue Knoten wählt hierbei Buddies aus, die der Nachbarschaftsmenge des Pastry-Netzwerkes angehören. Hierbei handelt es sich um die Knoten, die während des Einbindens des Knotens die beste vorhandene Überdeckungsrate, unabhängig von ihrer Netzwerkdistanz, aufwiesen.

Das Backup Protokoll. Jeder Pastiche Knoten kann selbst bestimmen, welche Daten, wann und wie oft gesichert werden sollen. Für die Verwaltung seines individuellen Archivierungsplans, baut jeder Knoten ein so genanntes „Metadaten-Skeleton“ für jedes durchgeführte Backup auf. Die Skeletons aller erhaltenen Backupzustände wird als eine Ansammlung persistenter, dateispezifischer Logs gespeichert, wie dies in den folgenden Abbildungen 8 und 9 schematisch dargestellt wird.

Abb. 8 beschreibt hierbei das Speicherformat eines Metadaten Chunks. Der Metadaten Chunk ist als eine Logdatei von Zuständen einer zu beschreibenden Datei bzw. eines zu beschreibenden Verzeichnisses organisiert. Jeder Eintrag der Logdatei beschreibt hierbei einen Zustand der Datei bzw. des Verzeichnisses nach einem

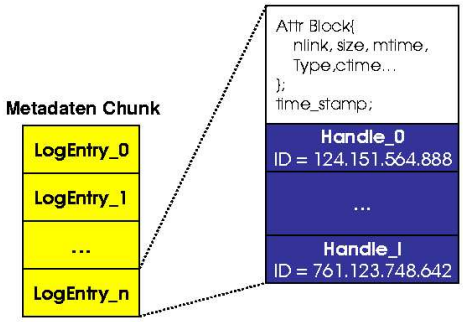


ABBILDUNG 8. Schematische Darstellung eines Metadaten Chunks

entsprechenden Update. Jeder Eintrag besteht aus einem Attributblock, einem Zeitstempel und einem Block von Chunk-Handles, aus der die jeweilige Momentaufnahme der Datei, die im Log beschrieben wird, zusammengesetzt ist.

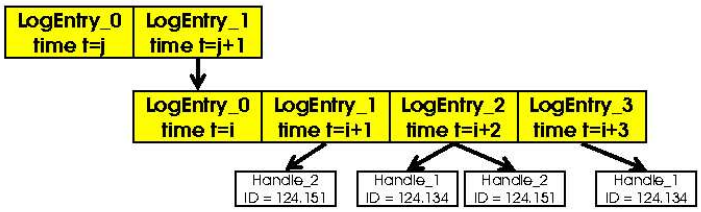


ABBILDUNG 9. Aufbau eines Chunk Skeletons

Zum Aufbau der Verzeichnisstruktur eines Systems verwendet Pastiche ein so genanntes Metadaten Skeleton. Abbildung 9 stellt den Aufbau eines solchen Skeletons dar. Hierbei wird jedes Element der Hierarchie als ein Logeintrag gespeichert. Jeder Logeintrag repräsentiert den Zustand des Elements zu einem gewissen Zeitpunkt, wird durch eine Menge von Chunk-Handles beschrieben (vgl. Abb. 8) und kann Pointer zu anderen Logeinträgen enthalten. Durch die Referenzen zwischen den Logeinträgen wird eine Hierarchie gebildet. Abb. 9 wurde durch verschiedene Vorgänge während des Backupprozesses aufgebaut. Zum Zeitpunkt t = j besteht das Skeleton nur aus einem LogEntry_0. Zum Zeitpunkt t = j+1 wird in der obersten Ebene ein Element hinzugefügt, z.B. durch das Anlegen eines neuen Verzeichnisses. Der neue Eintrag verweist auf einen neuen Logeintrag (t = i), der z.B. eine Datei in diesem Verzeichnis in Form eines Metadatenchunks (vgl. Abb. 8) repräsentieren könnte. Die Logeinträge zu den Zeiten t = i+1 bis t = i+3 entsprechen nun Updates, die durch Änderungen an dem durch Logeintrag 0 beschriebenen Element entstehen. So wird zum Zeitpunkt t = i+1 ein neuer Chunk-Handle_2 eingefügt. Zum Zeitpunkt t = i+2 wird das Element um einen weiteren Chunk-Handle_1 ergänzt, bevor der Chunk-Handle_2 zum Zeitpunkt t = i+3 wieder gelöscht wird.

Sowohl das Skeleton, das den aktuellen Zustand des System beschreibt, als auch alle durch die Backup Prozesse erhaltenen Momentaufnahmen des Systems, werden jeweils lokal und auf jedem Backup Knoten gespeichert.

Um eine erneute Sicherung der Daten durchzuführen, benötigt man die folgenden Informationen:

- Liste von Chunks, die zum Backup-Speicher hinzugefügt werden müssen (*„Add-Menge“*)

- Liste von Chunks, die vom Backup-Speicher gelöscht werden müssen (*„Delete-Menge“*)
- Liste von Metadatenobjekte des Sekeletons, die aufgrund der Änderungen angepasst werden müssen (*„Metadaten-Liste“*).

Zu Beginn der Durchführung eines Backups wird der öffentliche Schlüssel des Systems an alle Buddies versendet, da dieser dazu dient spätere Anforderungen zum Löschen oder Ersetzen von Chunks zu verifizieren. Danach werden die Chunk-IDs der Elemente in der Add-Menge gesendet. Sind hierbei Chunks auf dem entsprechenden Buddy noch nicht vorhanden, werden diese Daten ebenfalls an den entsprechenden Buddy weitergeleitet. Als nächstes werden die Elemente der Delete-Menge gesendet, wobei es sich hierbei nur um die Elemente handelt, die der Knoten in keinem seiner Backup Zustände mehr benötigt. Die Elemente der Delete-Liste müssen signiert sein, um böswilliges Löschen Dritter zu vermeiden.

Am Ende werden nun alle veränderten Metadaten Chunks gesendet. Da hierbei meist alte Chunks überschreiben werden, müssen auch alle Elemente der Metadaten-Liste signiert sein.

Nachdem alle Daten übertragen wurden, der Buddy die persistente Speicherung aller Elemente überprüft hat und der Vorgang für erfolgreich befunden wurde, sendet er eine Bestätigung zum Initiator des Backups zurück.

Bei diesem Verfahren, kann der Prozess zur Erstellung der Backupdaten stets wieder neu aufgesetzt werden, so dass durch etwaige Ausfälle der Netzwerkverbindung oder ähnlichem nicht wieder von Anfang an gestartet werden muss. Um im oben beschriebenen Prozess eine ausreichend gutes Leistung zu erzielen, wurden in Pastiche unterschiedliche Optimierungsverfahren integriert, auf die ich hier jedoch nicht näher eingehen werde.

Wiederherstellung eines Systems. Jeder Pastiche Knoten enthält eine lokale Instanz seines Archivierungs-Skeleton, so dass eine partielle Wiederherstellung des Systems relativ einfach ist. Der Knoten muss hierzu nur feststellen, welche Chunks für die Wiederherstellung der verlorengegangenen Dateien/Verzeichnisse benötigt werden und besorgt sich diese vom nächstgelegenen Buddy.

Im Falle eines Totalverlustes oder eines partiellen Ausfalls von dem auch das Skeleton betroffen ist, muss dieses zuerst wiederbeschafft werden. Hierzu wird auf jedem Buddy auch eine verschlüsselte Kopie des Wurzel-Metaobjekts des Knotens abgelegt. Beim Login des zu restaurierende Knotens in das Pastiche Netz wird diesem wieder seine ursprüngliche Knoten-ID zugewiesen, da diese aus dem Computernamen oder dessen IP berechnet wird. Daraufhin macht er sich auf die zufällige Suche nach einem Buddy der sein Wurzelobjekt enthält. Hat er einen solchen gefunden, so wird das Wurzelobjekt heruntergeladen und mit dem privaten Schlüssel entschlüsselt. Da dieses Wurzelarchiv nun Verweise auf alle Metadatenchunks enthält, kann das komplette System wiederhergestellt werden.

4.3. Zusammenfassung. Neben den in dieser Ausarbeitung behandelten Grundkonzepten, werden noch eine Vielzahl verschiedener Mechanismen zur Absicherung der Zusammenarbeit zwischen den Pastiche Knoten benötigt.

Zusammengefasst kann jedoch gesagt werden, dass Pastiche ein Verfahren ist, welches automatische Erstellung von Backups kostengünstig (durch Verwendung kostenlosen Speicherplatzes auf kooperierenden Endsystemen) ermöglicht und somit das Erstellen von Backups für den Anwender attraktiver macht. Außerdem wird durch die nur einmalige Speicherung gemeinsamer Daten, realisiert durch die Suche nach Buddies mit einer maximalen Datenüberdeckung, der benötigte Speicherplatz für das Backup minimiert.

Prototypen von Pastiche haben gezeigt, dass die zusätzlichen Operationen, die für das Erzeugen des Backups im Pastiche System notwendig sind, nur moderaten Overhead erzeugt.

Für genauere Details der technischen Konzepte, Implementierung und Erfahrungswerten im Praxiseinsatz sei an dieser Stelle auf [**CMN**] verwiesen.

Literaturverzeichnis

[ABKM02] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks, 2002.

[AT] Stephanos Androutsellis-Theotokis. A survey of peer-to-peer file sharing technologies (white paper).

[CGM] Arturo Crespo and Hector Garcia-Molina. Semantic overlay networks for p2p systems.

[CMN] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: Making backup cheap and easy.

[CSWH] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system.

[Gel] David Gelernter. The second coming ? a manifesto.

[Jov00] MA. Jovanovic. Modelling large-scale peer-to-peer networks and a case study of gnutella, June 2000.

[Kab] Mikhail Kabanov. Gnutella network snapshot.

[O'R] Tim O'Reilly. Peer to peer oder das netz als computer.

[Pat02] Frank Patalong. Kazaa-downloads: World-wide-weltmeister, July 2002.

[RDa] Antony Rowstron and Peter Druschel. Past: A large-scale, persistent peer-to-peer storage utility.

[RDb] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems.

[Shi00] Clay Shirky. What is p2p...and what isńt, November 2000.

KAPITEL 4

Internet Wide Storage – Ein Einblick in Moderne Verteilte Dateisysteme

Seminarbeitrag von **Tobias Kluge**

1. Motivation

Als „Internet Wide Storage“ werden Speichersysteme bezeichnet, die global verteilt sind, über das Internet oder ein anderes Netzwerk kommunizieren und aus mehreren tausend Rechnern bestehen können.

In dieser Ausarbeitung werden verschiedene Ansätze vorgestellt, die es sich zum Ziel gemacht haben, ein großes, skalierbares, sicheres und hoch verfügbares Speichersystem zu entwickeln. Aber was verbirgt sich dahinter?

In Zeiten des Internets liegt es nahe, Dateien auf Internet-Servern abzuspeichern. Im einfachsten Fall ist das ein Server, z.B. ein Webserver. Aber das ist nicht sehr effektiv. Durch die schnelle Verbreitung von Peer-to-Peer-Systemen (v.a. zum Tauschen von Dateien) wurde die Entwicklung von Peer-to-Peer-Dateisystemen angestoßen und schnell vorangetrieben. Die einzelnen Peers = Knoten stellen jeweils Speicherplatz zur Verfügung, um Dateien abzuspeichern. Das Problem dabei ist, einen möglichst guten Algorithmus zu entwickeln, um die Dateien effektiv zu verteilen und wiederzufinden. Außerdem ist das Internet ein beliebtes Feld für Hacker und Cracker, deswegen müssen Kommunikation sowie möglichst auch die Daten verschlüsselt werden. Erhöhte Anforderungen werden durch plötzliche Einwirkungen von Außen – wie dem Abbruch der Internet-Verbindung oder dem Ausfall von mehreren Knoten – gestellt, die im lokalen Netzwerk nicht in diesem Maß auftreten. In die Kategorie der Peer-to-Peer-Speichersysteme gehören vor allem Ivy und Pond.

Für große Netzwerke stellt Farsite eine geeignetes Dateisystem zur Verfügung, das die Last und den benötigten Speicherplatz automatisch auf alle Clients verteilt.

Past wird als „ultimativer Internet-Datenspeicher“ entwickelt. Es ist kein reines Peer-to-Peer-System, besteht aber trotzdem aus verschiedenen Maschinen, die zusammengeschaltet werden.

Zur Gliederung dieses Dokumentes: In Kapitel 2 werden die einzelnen Dateisysteme vorgestellt, insbesondere die Architektur und das Speichern und Finden von Dateien. Kapitel 3 geht auf spezielle Probleme ein und stellt die Lösungen der 4 Kandidaten vor. Sicherheit wird in Kapitel 4 getrennt betrachtet. Ein abschließender Überblick und der Versuch einer Bewertung erfolgt in Kapitel 5.

2. Überblick

In diesem Kapitel wird der Aufbau der Systeme vorgestellt und ein Einblick in den Ablauf beim Speichern und Finden von Dateien gewährt.

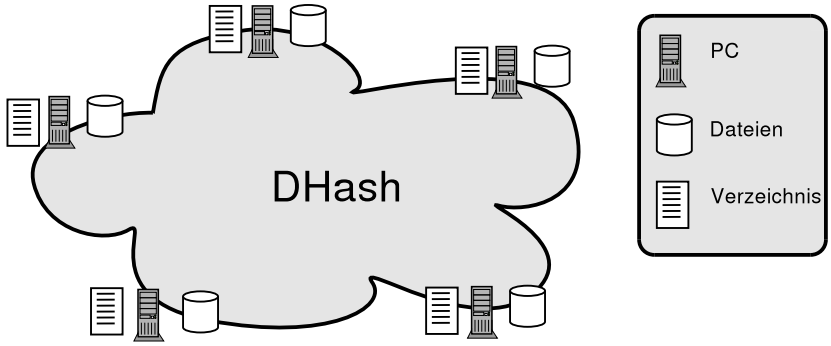
2.1. Architektur. Es gibt 2 verschiedene Ansätze – der klassische Client/Server-Ansatz (C/S) und das relativ junge Peer-to-Peer (kurz: P2P; es gibt keine zentralen Komponenten, alle Teilnehmer haben die gleichen Rechte und Pflichten; genaueres in [**Sib03**]) – die als Grundlage für die Entwicklung von Speichersystemen dienen. Allerdings gibt es auch Mischformen, die die Vorteile beider Architekturen nutzen.

Ivy. ist ein typisches Peer-to-Peer-Dateisystem. Alle Teilnehmer sind gleichberechtigt, jeder Knoten führt ein Logbuch – eine DHash-Tabelle; das ist eine verteilte P2P-Hash-Tabelle, die Schlüssel auf ihre entsprechenden Werte mappt. Die Integrität von Schlüssel/Wert-Paaren – sogenannten Blöcken – wird durch 2 Methoden sichergestellt: „content-hash block“, dabei ist der Schlüssel des Blocks ein SHA-1-Hash über den Dateiinhalt oder mittels „public-key block“, dabei ist der Schlüssel ein öffentlicher Schlüssel und die Datei muss mit diesem Schlüssel signiert sein. Zum Einfügen muss eine von diesen beiden Methoden verwendet werden, andernfalls wird die Änderung von Ivy zurückgewiesen.

Knoten schließen sich zu einer Gruppe, „view“ genannt, zusammen, die eine gemeinsame DHash-Tabelle nutzen und so ein gemeinsames Dateisystem bilden. Die Kommunikation der Knoten erfolgt über Chord (für weitere Informationen siehe: [**SMLN$^+$02**]), einem Routing-Protokoll für P2P-Systeme. Die einzige direkte Kommunikation erfolgt beim Erstellen der Gruppe.

Ivy kann wie ein ganz normales NFS-System gemounted werden; für den Nutzer ist es dabei nicht ersichtlich, dass es sich um ein verteiltes Peer-to-Peer-Speichersystem handelt.

Abbildung 1. Ivy: gleichberechtigte Peer-to-Peer Knoten, die Chord kommunizieren

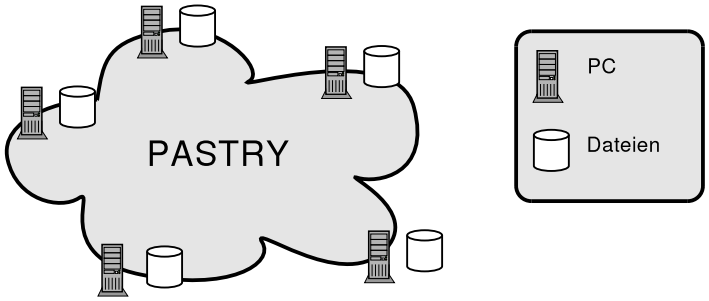


Past. ist ebenfalls ein Peer-to-Peer-Dateisystem. Im Gegensatz zu Ivy ist es aber nicht möglich, Dateien zu bearbeiten bzw. zu ändern; für jede geänderte Datei wird eine neue Datei in das System eingefügt (weitere Informationen dazu im Abschnitt Speichern von Dateien).

Jeder Knoten stellt Speicherplatz zur Verfügung, dieser wird für das Speichern von Daten aus dem Speichernetzwerk sowie für das Zwischenspeichern von Anfragen verwendet.

Das Routing von Nachrichten erfolgt durch Pastry (siehe Past im Abschnitt Speichern von Dateien sowie in Literatur [**RD01**], [**Sib03**]). Ebenfalls durch Pastry können Dateien lokalisiert werden, d.h. herausgefunden werden, wo sie gespeichert sind.

Abbildung 2. Past: gleichberechtigte Knoten mit lokalem Speicher, die über Pastry kommunizieren



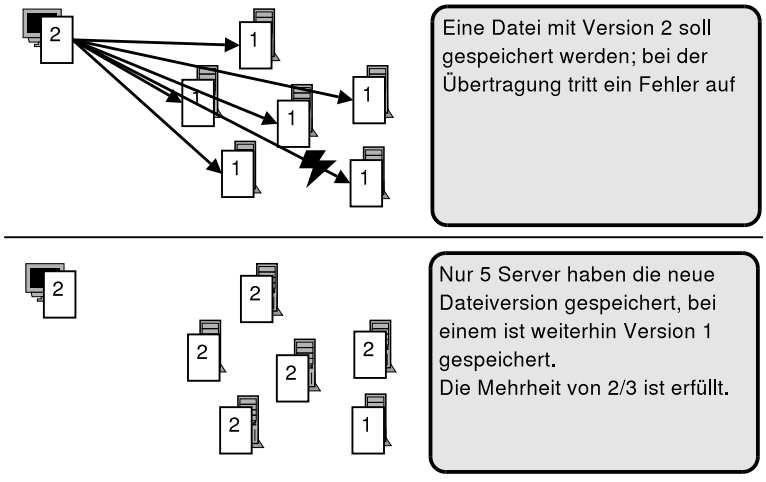
Farsite. Von Außen gesehen ist Farsite ein typisches Client/Server-Dateisystem mit einem zentralen Dateiserver. Allerdings verbirgt sich unter der Haube des Servers ein Netzwerk von Workstations, die zu Gruppen zusammengeschlossen sind und die Dateien gemeinsam verwalten.

Dabei gibt es normale Clients, die beliebig an- und ausgeschaltet werden können, und spezielle Dateiserver, auf denen die Dateien abgelegt werden. Das geschieht mittels Byzantine-Protokoll; dabei wird sichergestellt, das Änderungen an einer Datei nach dem Speichern auf mehreren Servern bei Anfragen nach dieser Datei immer die aktuellste Version liefern. Im Folgenden wird es kurz vorgestellt:

Beim Speichern bekommt der Client von jedem Server eine Bestätigungsnachricht (die meist signiert ist, um den Server verifizieren zu können). Haben mehr als $\frac{2}{3}$ der Server mit einer Bestätigung geantwortet, ist die Datei für den Client erfolgreich gespeichert. Fragt jetzt ein Client an, dann bekommt er von allen Server die Datei; stimmen mehr als 50% einer Version überein, dann kann er sicher sein, dass er die aktuelle, richtige Version erhalten hat.

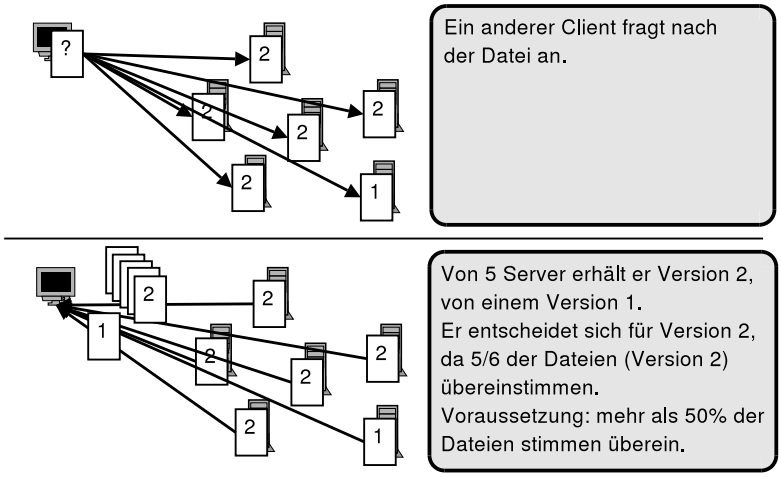
Ein Beispiel: Es gibt 6 Server, die alle die gleichen Dateien speichern. Ein Client speichert eine Datei auf allen 6 Servern, bekommt 5 Bestätigungen, die Datei ist damit erfolgreich gespeichert.

Abbildung 3. Speichern beim Byzantine-Protokoll: 5 von 6 Servern haben neue Version erfolgreich gespeichert



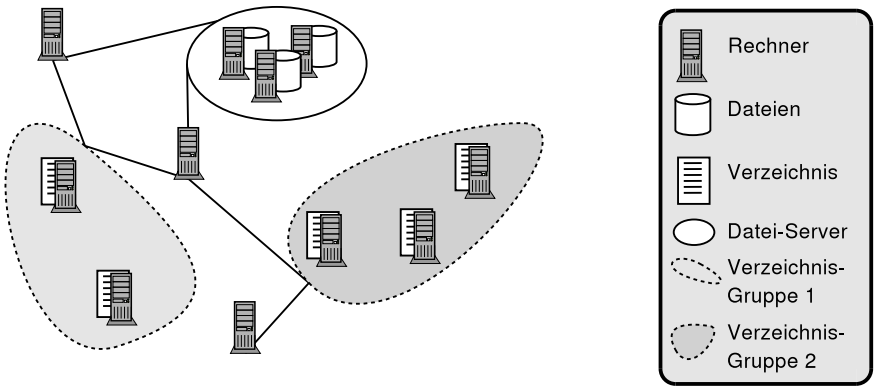
Die Anfrage eines anderen Clients liefert 5 neue Dateien und eine veraltete Datei zurück – mehr als 50% der Dateien stimmen überein, er weiß damit, dass die 5 Dateien in der richtigen, aktuellen Version vorliegen.

Abbildung 4. Anfordern einer Datei beim Byzantine-Protokoll: 5 von 6 Dateien stimmen überein, damit wird diese Datei verwendet



Die Verzeichnisse sind auf Gruppen, verschiedene normale Client-PCs, verteilt. Es gibt eine Wurzelgruppe, in der das Dateisystem beginnt – ähnlich wie in Unix das „/“. Jeder Rechner der Gruppe speichert die gleichen Informationen. Werden in dieser Gruppe die Ressourcen knapp, wird eine neue Gruppe gebildet und ihr einen Teil der Verzeichnisstruktur übertragen. Für diese neue Gruppe werden mehrere PCs zufällig aus allen verfügbaren ausgewählt. In der Verzeichnistabelle der Wurzelgruppe wird anstelle der Verzeichnisstruktur ein Link auf die neue Gruppe eingetragen; Anfragen nach Informationen zu diesen Verzeichnissen werden an die neue Gruppe weitergeleitet.

Abbildung 5. Farsite: Verzeichnis-Gruppen, Datei-Server, einzelne Clients

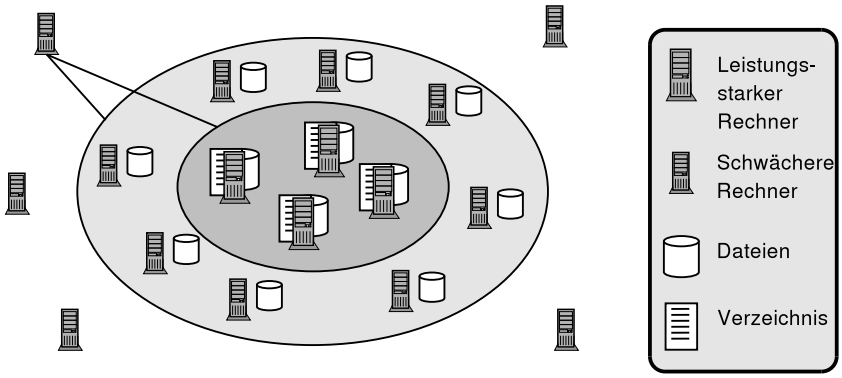


Pond. Pond besteht aus einem „Inneren Ring“ – starken Servern mit sehr guter Netzwerkanbindung – und einem „Äußeren Ring“ – wechselnden, schwächeren PCs mit weniger guter Netzwerkverbindung. Die Anfrage wird dabei an den äußeren Ring gestellt, und wenn sie von diesem nicht beantwortet werden kann, an den

inneren Ring weitergeleitet. Dabei wird für die „Ringkommunikation“, Antworten des inneren Rings an den äußeren, ebenfalls das Byzantine-Protokoll verwendet.

Das Routing von Nachrichten und Finden von Objekten wird von Tapestry erledigt; das funktioniert ähnlich wie Pastry, genauere Informationen können [**ZKJ01**] entnommen werden.

Abbildung 6. Pond: Innerer Ring, Äußerer Ring, Clients.

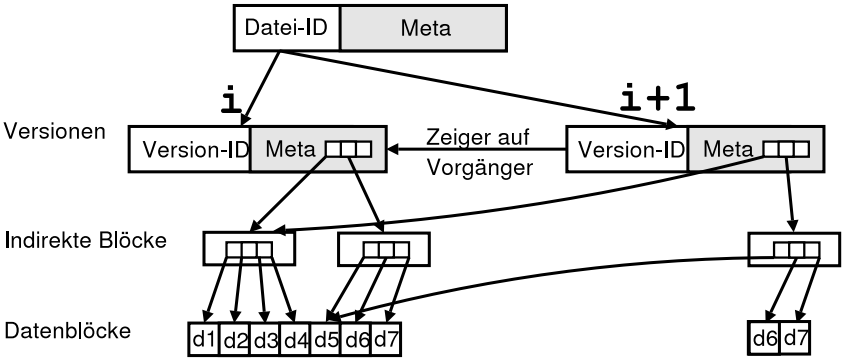


2.2. Speichern von Dateien. Dateien werden in Blöcken aufgeteilt gespeichert. Dabei können diese Blöcke auf unterschiedlichen Rechnern abgespeichert werden, bei den hier vorgestellten Systemen z.B. bei Pond.

Ein Knackpunkt beim Speichern sind sich überschneidende Änderungen von Dateien. Wird eine Modifikation von Dateien zugelassen gibt es Probleme mit dem Caching und der Konsistenz, genaueres dazu in Kapitel 3. Entweder wird deswegen das Ändern von Dateien ausgeschlossen (Past), für jede Änderung eine neue Version angelegt (Pond), Rechte zum Ändern nur für ein begrenztes Zeitfenster gestattet (Farsite) oder ein Logbuch über durchgeführte Änderungen gepflegt (Ivy).

Für das Speichern von verschiedenen Versionen einer Datei wird z.B. bei Pond folgendes Verfahren eingesetzt: Die Datei wird mit einem Schlüssel oder einer ID identifiziert, mit der auf einen Datei-Kopf verwiesen wird; dieser zeigt auf die neuste Version. Die verschiedenen Versionen werden in einer verketteten Liste gespeichert, in dieser Liste kann von der neusten Version zu einer älteren durchgegangen werden. Meist werden Dateiinformationen wie Besitzer, Zugriffsinformationen oder letzte Änderung im Kopf mit abgespeichert.

Abbildung 7. Datei in Pond



Ivy. Dateien können hinzugefügt, gelesen, geändert und gelöscht werden.

Beim Einfügen wird die Datei auf dem Knoten selbst gespeichert und ein Eintrag in das lokale Logbuch erstellt. Dadurch – die Änderungen im lokalen Logbuch sind sofort für alle anderen Gruppenmitglieder sichtbar – ist die Datei für alle anderen Gruppenmitglieder lesbar. Alle Änderungen an dieser Datei werden ebenfalls im Logbuch protokolliert. Im Log selbst wird für jede Datei eine verkettete Liste von schreibgeschützten Logeinträgen verwaltet, der Kopf der Liste zeigt auf die aktuellste Version.

Damit ist es möglich, frühere Versionen von Dateien herzustellen. Ebenso lassen sich Änderungen leicht verfolgen.

Probleme treten allerdings auf, wenn einzelne Knoten vom Netz verschwinden: dann ist es nicht mehr möglich, auf die Dateien, die auf dem jeweiligem Knoten gespeichert sind, zuzugreifen. Es wird ein Tool mitgeliefert, dass Verweise auf fehlende Dateien entdeckt und versucht, diese sinnvoll zu ersetzen.

Past. In Past wird Pastry als Routing-Protokoll verwendet. Jeder Knoten und jede Datei besitzen dabei eine eindeutige ID. Die Dateien werden auf den Knoten abgespeichert, deren Knoten-ID am wenigsten von der der Datei-ID abweicht. Beim Speichern kann angegeben werden, wie viele Kopien (k) von der Datei angelegt werden sollen. Können nicht alle k Knoten die Datei speichern, weil z.B. zu wenig Speicherplatz auf den Knoten vorhanden ist, besteht die Möglichkeit einer „file diversion“. Dabei wird die Datei-ID neu berechnet und der Speichervorgang beginnt von vorn. Dies kann bis zu 4 mal geschehen, danach schlägt die Speicherung der Datei fehlt und die Anwendung wird davon informiert.

Die Datei-ID wird aus einem Hash über den Dateinamen, den öffentlichen Schlüssel des Besitzers und einem „random salt“[1] berechnet.

Wird auf einem Knoten der Speicherplatz knapp, kann er Dateien auslagern. Dazu sucht er in seiner Routing-Tabelle nach einem Knoten und sendet ihm eine Nachricht. Kann dieser Knoten die Datei speichern, wird sie zu dem Knoten transferiert und im Verzeichnis wird ein Verweis auf den entsprechenden Knoten abgespeichert. Dieser Vorgang wird „replica diversion“ genannt.

Farsite. Für das Speichern einer Datei muss der Client bei der Verzeichnisgruppe nach einer Berechtigung anfragen. Diese prüft anhand einer Liste von Clients, die Dateien speichern dürfen, ob er dazu berechtigt ist (wird als Acess Control List (ACL) bezeichnet). Wenn ja, wird dem Client ein Zertifikat ausgestellt, mit dem er die Datei auf den Dateiservern abspeichern darf. Die Kommunikation der Clients mit den Dateiservern erfolgt nach den Regeln des Byzantine-Protokoll.

Für das Hinzufügen von neuen Dateien muss der Client am System autorisiert sein, dass heißt, einen gültigen Account besitzen.

Das Löschen von Dateien ist nur für schreibberechtigte Nutzer möglich.

Pond. Dateien werden in Daten-Objekten abgespeichert; diese bestehen aus verschiedenen schreibgeschützen Versionen der Datei. Identifiziert wird es mittels einer ID, die mittels einer Hash-Funktion aus Dateiname und öffentlichem Schlüssel des Besitzers erzeugt wird. Mit ihr wird auf die Versionen der Datei zugegriffen. Die Versionen selbst sind als schreibgeschützte B-Bäume abgespeichert – die Blätter beinhalten dabei jeweils die Datenblöcke.

Änderungen werden an den inneren Ring geschickt. Dort werden sie mittels Byzantine-Protokoll übernommen und die geänderten Daten an den äußeren Ring gesendet, der diese ggf. an angebundene Clients weiterleitet.

In Pond werden „*erasure codes*“ verwendet, um die Dauerhaftigkeit von Datei-Blöcken abzusichern.

[1] Das ist eine „Würze“ die dazu dient, den Hash so zu variieren das verschiedene Datei-IDs für die gleiche Datei erzeugt werden können.

Dabei wird der Block in m gleich große Fragmente aufgeteilt, die in n Fragmente umkodiert werden (n ¿ m). Der Original-Block kann dann aus nur m (von n) Fragmenten wiederhergestellt werden. Die Kodierrate r berechnet sich aus $r = \frac{m}{n} < 1$, die Kosten für die zusätzliche Speicherung steigen um Faktor $\frac{1}{r}$.

Beim Speichern oder Updaten von Blöcken in Pond werden diese mit erasure codes kodiert und die Fragmente auf dem inneren Ring abgespeichert, der Zugriff erfolgt mittels Tapestry. Bei der Rekonstruktion werden die entsprechenden Fragmente (m sind ausreichend) mittels Tapestry lokalisiert und anschließend rekodiert.

Die zusätzliche Langzeit-Sicherheit wird durch eine längere Laufzeit beim Abspeichern und Lesen von Dateien erkauft. Das einmalige Kodieren einer neuen oder veränderten Datei übernimmt der innere Ring. Das Enkodieren hingegen wird an den äußeren Ring delegiert: er fordert kodierte Dateien vom inneren Ring an, berechnet die Datei daraus und liefert sie an Clients aus.

2.3. Finden von Daten. Es gibt verschiedene Möglichkeiten, auf Dateien zuzugreifen. Die Verweise auf Dateien können in einem Verzeichnis gespeichert werden. Einige Systeme setzen voraus, dass der Nutzer die ID der Datei kennt – wobei dem Nutzer kein Verzeichnis zur Verfügung gestellt wird, er sich also selbst darum kümmern muss, die eine Datei die entsprechende ID zu finden.

Ivy. Durch den Log können Dateien gefunden werden, er fungiert als eine Art „Directory“. Ivy durchsucht alle Logeinträge in $log(n)$ und gibt, wenn er zuerst auf ein „Unlink“ – d.h. einen Lösch-Eintrag – stößt, zurück, dass die Datei nicht existiert. Findet er normale Logeinträge zur Datei, gibt er die aktuellste Version zurück.

Das Routing der Nachrichten und Finden von Dateien geschieht mittels Chord, das ähnlich wie Pastry (nächster Abschnitt) DateiIDs auf bestimmte HostIDs mappt. Genauere Informationen sind unter [**SMLN$^+$02**] zu finden.

Past. Zum Finden von Dateien wird die Datei-ID benötigt, an einem Verzeichnisdienst wird noch geforscht. Es wird eine Anfrage an das System gestellt, die an den Knoten weitergeleitet wird, dessen Knoten-ID am wenigsten von der Datei-ID abweicht. Kann ein Knoten auf dem Weg zum Zielknoten die Anfrage bearbeiten, sendet er die Antwort und verwirft die Anfrage. Andernfalls wird die Anfrage bis zum Zielknoten weitergeleitet.

Wurde die Datei auf einen anderen Knoten ausgelagert, schickt der Original-Speicher-Knoten die Anfrage an den entsprechenden Knoten weiter, der dann auf die Anfrage antwortet.

Farsite. Alle Datei-Informationen (Name, weitere Metadaten) werden in einem Directory-Verzeichnis gespeichert, allerdings verschlüsselt (außer Dateiname) – mit den öffentlichen Schlüsseln der Clients, die später darauf zugreifen dürfen.

Das Verzeichnis bildet einen Namensraum mit einer Wurzel („Root“). Unternamensräume können an weitere Gruppen delegiert werden.

Der Client selbst fragt beim Wurzel-Namensraum an, hangelt sich anschließend bis zur Datei durch und bekommt von der verwaltenden Gruppe die Meta-Informationen. Diese muss der Client mit seinem geheimen Schlüssel entschlüsseln, darin sind u.a. auch die Namen der Dateiserver enthalten. Bei diesen fragt er nach der Datei an und bekommt die verschlüsselten Blöcke, die anschließend entschlüsselt werden.

Pond. Pond selbst stellt kein Verzeichnis zur Verfügung, das heißt, die Clients müssen selbst die Verwaltung der Datei-IDs übernehmen.

Mit der Datei-ID ausgerüstet fragen sie den äußeren Ring. Dieser speichert einen Teil der Dateien lokal; bei Dateien, die nicht gespeichert werden, wird eine Anfrage an den inneren Ring erstellt, der die Datei zurückliefert. Dieser Vorgang wird im Abschnitt Caching detailiert behandelt.

Das Versenden der Nachrichten wird von Tapestry erledigt.

3. Techniken

In diesem Kapitel werden verschieden Verfahren vorgestellt, die für ein internetweites Dateisystem notwendig sind. Dabei geht es besonders um die Beschleunigung von Anfragen durch Caching und redundaten Speichern durch Replikation.

3.1. Caching. Caching beschreibt das Verwalten von Dateien, die aus Performance-Gründen zwischengespeichert, aber gleichzeitig auch bearbeitet (lesend, schreibend) werden können.

Meist werden temporäre Änderungen lokal zwischengespeichert und erst nach einer Zeitspanne gesammelt an die speichernde(n) Maschine(n) übermittelt. Das dabei Probleme auftreten, ist offensichtlich: Bearbeitet ein Benutzer eine lokale Kopie, während ein anderer die veraltete Version vom Server läd und ebenfalls bearbeitet, tritt eine Inkonsistenz auf.

Verschiedene Lösungsansätze, die in den Systeme verwendet wurden, werden auf den folgenden Seiten diskutiert. Für eine tiefergehende Diskussion kann z.B. [**Tan95**] verwendet werden.

Read-Only-Dateien. Die einfachste Lösung ist die, Dateien nicht bearbeiten zu lassen. Wird eine Datei geändert, wird sie als neue Datei angelegt.

Dieses Prinzip verwendet Past. Geänderte Dateien erhalten eine neue Datei-ID, die Originaldatei bleibt unangetastet.

Klassisches Caching. Dateien dürfen bearbeitet werden. Die Speicherung kann auf dem Server und auf dem Client geschehen.

Die Konsistenzprobleme löst Farsite, indem es dem schreibenden Nutzer nur eine bestimmte Zeit zugesteht. In diesem Zeitraum kann die Datei beliebig verändert werden. Fragt ein anderer Nutzer nach dieser Datei, wird dem schreibenden eine Nachricht geschickt und ein Ultimatum gestellt. Wenn die Datei verändert wurde, schickt der änderne Nutzer sie an den Server. Anschließend kann die Datei vom Server heruntergeladen werden.

Eine weitere Verbesserung wird dadurch erzielt, dass die Änderungen des Nutzers gepuffert werden, d.h. in einem bestimmten Intervall an den Server geschickt werden. Das ist insbesondere vorteilhaft für kurzlebige Dateien, die erzeugt und kurz darauf wieder gelöscht werden, da diese Änderungen nicht zum Server geschickt werden.

Log in Ivy. Ivy verwendet Logs, in denen die Änderungen an Dateien protokoliert werden. Diese Logs sind global verfügbar, Änderungen sind sofort bei den Clients sichtbar. Das funktioniert allerdings nur, wenn das unterliegende Peer-to-Peer-Netzwerk nicht geteilt wurde.

Außerdem besitzt jeder Knoten einen Schnappschuss des Systems. Das ist eine DHash-Tabelle, in der alle Datei-IDs abgespeichert sind. Änderungen am System werden hier nachvollzogen.

Wurde die Gruppe geteilt und später wieder zusammengeführt, kann des zu dem Problem führen, dass einige Dateien von verschiedenen Nutzern gleichzeitig verändert wurden, die Änderungen sich aber überschneiden. Dafür stellt Ivy ein Tool zur Verfügung, das manuell von Hand gestartet werden muss. Es erstellt eine Übersicht der sich überschneidenden Änderungen vor der Teilung der Gruppe und eine für jede geteilte Gruppe. Allerdings muss der Nutzer entscheiden, welche Variante er weiter nutzen möchte. Zusammenführen von beiden Änderungen muss er selbst durchführen – im Paper wird es folgendermaßen beschrieben:

> „[...] merge them by hand in a text editor.“ [**MMGC02**, Seite 8] – der Nutzer muss es die Änderungen selbst im Text-Editor zusammenführen.

Schichten. Pond speichert Dateien im inneren Ring. Dabei werden sie aber in einem bestimmten Format abgelegt (siehe Abschnitt Ereasure Codes), aus dem

in einem aufwändigen Verfahren die Datei extrahiert werden muss. Bei Anfragen an Dateien wird zuerst der äußere Ring abgefragt, der bereits entpackte Dateien speichert. Wenn von ihm die Anfrage nicht beantwortet werden kann, fragt er beim inneren Ring nach der gepackten Version der Datei an, entpackt sie, speichert sie lokal ab und sendet die entpackte Datei an den Client. Bei der nächsten Anfrage kann er direkt eine Antwort geben, der innere Ring wird entlastet.

Beim Änderungen an einer Datei werden die Rechnern im äußeren Ring, die eine entpackte Version besitzen, informiert. Diese können dann die aktuelle Version anfragen und verarbeiten.

3.2. Replikation. Darunter wird das mehrfache Speichern einer Datei auf verschiedenen Computern verstanden.

Ziel ist, den Zugriff auf eine Datei zu beschleunigen und beim Ausfall eines Speicherortes auf einen anderen ausweichen zu können. Das mehrfache Speichern einer Datei führt allerdings wieder zu Konsistenzproblemen.

Farsite. Farsite verwendet mehrere gleichberechtigte Dateiserver, auf denen die Dateien abgelegt werden.

Pond. Pond hingegen unterscheidet zwischen primärer und sekundärer Replikation. Die primäre hat die richtigen, immer aktuellen Daten, die Maschinen der sekundären halten zwischengespeicherte und vorverarbeitete Daten bereit, sie sind also ein Cache für Client-Anfragen. Änderungen werden durch Push von der primären Replikation an die sekundäre weitergeleitet.

Clients können verifizieren, dass sie die aktuellste Version einer Datei besitzen, indem sie eine „heart beat“-Anfrage an den inneren Ring schicken. Dabei senden sie ein beliebiges Wort mit in der Anfrage an die primären Replikationsserver und erhalten von diesem ein Zertifikat mit der Datei-ID, der Versionnummer, einem Zeitstempel, dem gesendeten Wort und dem Namen des anfragenden Clients, signiert mit dem Schlüssel der primären Replikation.

Ivy. Ivy ermöglicht Replikation über die verteilte Hash-Tabelle, dabei werden die Dateisysteminformationen automatisch auf allen Maschinen gespeichert, die Dateien selbst allerdings nicht.

Past. Bei Past wird beim Erzeugen angegeben, wie oft die Datei abgespeichert werden soll. Diese wird dann k-mal auf den k numerisch nächsten Knoten abgespeichert. Von diesen werden jeweils Bestätigungsnachrichten verschickt. Ist während der Laufzeit der Speicherplatz eines Knotens nicht ausreichend, kann der Knoten Dateien selbstständig auf einen anderen Knoten in seinen Tabellen verschieben; er selbst speichert in seiner Dateiliste einen Zeiger auf den Knoten, um Anfragen nach der Datei beantworten zu können. Dieser Vorgang wird „replica diversion“ bezeichnet.

4. Sicherheit

Bei der Kommunikation über das Internet oder selbst das LAN ist das System verschiedenen Angriffen ausgesetzt, z.B. das Abhören der Kommunikation, Einfügen von Phantom-Nachrichten oder Weglassen von Nachrichten, das Übernehmen von ganzen Hosts oder einfach dem Abbruch der Verbindung einer Maschine mit dem Rest des Systems.

Um die Risiken zu minimieren gibt es verschiedene Ansätze, die nachfolgend diskutiert werden.

4.1. Verschlüsselung. Zur Geheimhaltung der Kommunikation und der Dateien werden sowohl symmetrische als auch asymmetrische Verschlüsselungsverfahren eingesetzt.

Symmetrische Verfahren verwenden einen geheimen Schlüssel, der beiden Seiten bekannt sein muss. Mit diesem wird die Nachricht verschlüsselt. Ein bekannter

Vertreter ist DES. Sie werden oft zum Verschlüsseln von Dateien und größerer Nachrichten verwendet, da sie sehr schnell sind.

Bei asymmetrische Verfahren besitzt jeder Kommunikationspartner 2 Schlüssel: einen geheimen und einen öffentlichen. Dabei wird die Nachricht mit dem öffentlichen ver- und mit dem geheimen vom Empfänger entschlüsselt; bekannt ist z.B. RSA. Allerdings ist dieses Verfahren im Vergleich zur Verschlüsselung symmetrischen Schlüsseln langsamer.

In Pond und Farsite werden Zugriffsrechte dadurch garantiert, dass die symmetrischen Schlüssel (für die eigentlichen Daten) mit den öffentlichen Schlüsseln der Nutzer verschlüsselt werden. Andere Nutzer haben keine Chance, an die Daten zu kommen. Das Verfahren wird „convergent encryption“ genannt und im nächsten Abschnitt vorgestellt.

Allerdings ist die Verwendung von Verschlüsselung bei den hier vorgestellten Systemen sehr unterschiedlich. So verschlüsselt Farsite fast die gesamte Kommunikation und Daten, in Ivy und Past hingegen wird es dem Nutzer überlasssen, in wie weit er seine Daten schützen möchte.

4.2. Autentifizierung. Autentifizierung bezeichnet die eindeutige Identifikation des Nutzers am System.

Durch die Verwendung von Kryptografie wird dafür von den meisten Systemen ein asymmetrischer, signierter Schlüssel verwendet. Dabei wird vom System normalerweise keine Anforderung daran gestellt, wo und wie die Schlüssel abgespeichert werden – eine Ausnahme ist Past, da es die Möglichkeit bietet, Schlüssel auf Smartcards zu speichern.

Werden die Schlüssel auf der Festplatte gespeichert, kann sich jeder, der Zugriff – berechtigt oder unberechtigt – auf diesen Rechner und die Festplatte hat, als dieser Nutzer ausgeben. Also ist diese Authentifizierung nicht sicher.

Ivy. Jede Maschine besitzt einen asymmetrischen Schlüssel. Nutzer selbst werden über die Maschine identifiziert, dadurch ist keine garantierte Nutzerauthentifizierung möglich.

Past. Jeder Nutzer und jeder Knoten besitzt einen asymmetrischen Schlüssel, der auf einer Smartcard oder der Festplatte gespeichert wird; im Falle der Smartcard ist eine Nutzer-Authentifizierung garantiert – wenn der Zugriff auf die Smartcard nicht gehackt wurde.

Farsite. Nutzer und Maschinen werden mit asymmetrischen Schlüsseln authentifiziert. Diese werden allerdings auf den Festplatten der Rechner abgespeichert, sind also (theoretisch) nicht so gut geschützt.

Pond. Nutzer werden mittels ihrer asymmetrischen Schlüssel authentifiziert, die Maschinen des inneren und äußeren Rings ebenfalls. Die Rechner der Nutzer außerhalb der Ringe hingegen werden nicht authentifiziert.

4.3. Autorisierung. Mit Autorisierung wird die Berechtigung (keine, nur lesend, lesend und schreibend) von einzelnen Komponenten des Systems bezeichnet. Ziel ist es, Daten und Maschinen von Manipulation von Außen zu schützen.

Die Unterscheidung nach den verschiedenen Berechtigungen erfolgen nach einer erfolgreichen Autentifizierung.

Ivy. Jeder Client besitzt einen asymmetrischen Schlüssel.

Innerhalb der Gruppe darf jeder Client die Dateien lesen und Änderungen durchführen, er ist zu allem berechtigt. Wird ein Client übernommen, dann hat der Angreifer volle Zugrifs- und vor allem Änderungsrechte auf alle Dateien. Da die Änderungen aber protokolliert werden, können diese später zurückgenommen werden.

Wurde erkannt, dass ein Client übernommen wurde, wird – von den Nutzern per Hand initiiert – eine neue Gruppe ohne den befallenen Knoten gebildet. Hatte dieser Daten gespeichert, die sonst nirgendwo im Netz gespeichert werden, sind

diese verloren. Diese Inkonsistenz muss manuell behoben werden, dafür steht ein externes Tool bereit.

Past. Die Berechtigungen bei Past sind differenziert:

- Alle Nutzer dürfen Dateien lesen.
- Nur authentifizierte Nutzer dürfen Dateien hinzufügen.
- Nur der Besitzer einer Datei darf diese löschen.

Dabei wird die Berechtigung mittels asymmetrischer Schlüssel überprüft, der Inhalt der Datei aber – per default – nicht verschlüsselt.

Farsite. Farsite nutzt verschiedene Verfahren, um Lese- und Schreibrechte zu verwalten.

In den Meta-Daten des Datei-Verzeichnisses ist eine „Access Control List“ (ACL) abgelegt, in der alle öffentlichen Schlüssel der Nutzer gespeichert sind, die Dateien und Verzeichnisse ändern dürfen. Bei allen Änderungen wird der Schlüssel des Nutzers mit denen in der ACL verglichen und nur bei Übereinstimmung durchgeführt.

Die Dateien selbst werden mittels „convergent encryption“ verschlüsselt. Dabei wird die Datei in Blöcke aufgeteilt, für jeden Block jeweils ein Hash berechnet und mit diesem symmetrisch verschlüsselt. Die Block-Hashs selbst werden asymmetrisch mit allen öffentlichen Schlüsseln der Nutzer verschlüsselt, die später diese Datei lesen dürfen. Das hat den Vorteil, dass identische Dateien – unabhängig von den leseberechtigten Nutzern – immer gleich verschlüsselt werden; gleiche Dateien sind auch verschlüsselt als solche zu erkennen.

Datei- und Verzeichnisnamen werden ebenfalls mit einem symmetrischen Schlüssel, der mit den öffentlichen Schlüsseln der leseberechtigten Nutzern geschützt ist, verschlüsselt. Mit der Technik „exclusive encryption“ wird garantiert, das beim Entschlüsseln immer „legale“ Bezeichnungen produziert werden, unabhängig davon, was der Nutzer als Name eingegeben hat.

Pond. Der Zugriff auf eine Datei wird in lesend und schreibend unterteilt. Die Leseberechtigung wird durch eine Verschlüsselung mit den öffentlichen Schlüsseln der berechtigten Nutzer garantiert, Schreibberechtigung wird auf den Servern des Inneren Rings mittels Check gegen ACL gewährt.

Die Kommunikation zwischen den verschiedenen Ebenen ist sehr unterschiedlich:

Innerhalb des inneren Rings wird mittels des Byzantine-Protokoll kommuniziert, mit der Besonderheit, das die Nachrichten nicht asymmetrisch sondern symmetrisch mit Nachrichtenauthentifizierungsschlüsseln (MAC) verschlüsselt werden. Da für jede Verbindung zwischen 2 Maschinen ein eigener Schlüssel existiert, muss die Anzahl der Rechner im inneren Ring klein gehalten werden.

Der Rest der Kommunikation (innerer Ring mit äußerem, Nutzern sowie äußerem und Nutzer) wird mit asymmetrischer Verschlüsselung geschützt. Verbessert wird die Kommunikation mit dem inneren Ring durch Verwendung von „proactive threshold signatures“. Dabei bilden und vereinbaren mehrere Maschinen einen gemeinsamen Schlüssel. Wenn mehr als $\frac{2}{3}$der Maschinen übereinstimmen, kann eine gültige Signatur erzeugt werden. Weitere Informationen sind unter [**Rab98**] zu finden.

5. Zusammenfassung

Die Dateisysteme befinden sich noch in der Entwicklung, deswegen kann kein direkter Vergleich erfolgen. Außerdem sind die sehr Motivationen unterschiedlich.

Bezüglich Geschwindigkeit dient NFS (ein Netzwerk-Dateisystem, von Sun entwickelt; besteht aus einem zentralen Server und Clients und kann deswegen schlecht skalieren) als Referenzobjekt. Beim Lesen sind alle Kandidaten schneller (Pond: 4.6

mal schneller) oder zumindest nicht viel langsamer (Ivy: 18% langsamer bei 4 Knoten) wie NFS, allerdings brechen sie beim Schreiben ein. Ivy ist dann nur halb so schnell wie NFS, Pond benötigt die siebenfache Zeit. Für Past wurden leider keine Daten gefunden; Farsite wird in [**ABC**$^+$**02**] mit NTFS verglichen und benötigt die doppelte Zeit wie NTFS (außer beim Erzeugen von Verzeichnissen, da benötigt Farsite die Hälfte der Zeit). Die Gründe für die schlechten Schreib-Zeiten sind darin zu suchen, dass die Systeme für ein „schnelles“ Lesen optimiert sind; da die Dateien verteilt sind und z.T. kodiert werden müssen, benötigt das Speichern mehr Zeit. Ein weiterer Punkt ist, dass die Signallaufzeiten im Internet größer als im lokalen Netzwerk. Außerdem ist NFS stark optimiert, die hier vorgestellten Systeme befinden sich noch in der Entwicklung und sind nicht sehr optimiert.

Die Systeme sind für unterschiedliche Größen konzipiert: Past und Ivy werden eher für kleinere bis mittlere Systeme verwendet, Farsite für lokale Netzwerke mit maximal 10^5Rechnern, Pond für sehr große. Die Testsets, die für die Performance-Messung verwendet wurden, umfassten meist 20-100 Knoten, die global verteilt sind. Dafür wird von Pond und Ivy das von PlanetLab – einer globalen Testumgebung für das Entwickeln und Testen neuer Netzwerkdienste – bereitgestellte Computernetz verwendet. Die Knoten sind in Nord-Amerika, Europa, Australien und Neuseeland verteilt. Für Systeme mit mehreren Tausend Knoten liegen bei den hier behandelten Kandidaten bisher keine Erkenntnisse vor.

Bezüglich Sicherheit sind große Unterschiede sichtbar: die Peer-to-Peer-Systeme bieten diesbezüglich weniger als Farsite und Pond.

Alles in allem sind die Peer-to-Peer-Systeme nicht so leistungsfähig wie die mehr oder weniger zentralen Systeme Farsite und Pond. Die Dienste (Verzeichnis, Verschlüsselung und Authentifizierung von Dateien & -inhalten), die von ihnen bereit gestellt werden, müssen bei Past und Ivy nachgebildet werden. Allerdings ist die Entwicklung noch nicht abgeschlossen.

Ausblick. Die Möglichkeiten, die diese Dateisysteme bieten, sind sehr interessant. Auf diesem Gebiet wird sich in nächster Zukunft viel tun. In nicht ferner Zukunft wird sicherer und dauerhafter Speicherplatz in einem Internet-Dateisystem gemietet werden können. Die Kosten für Sicherungen lassen sich dadurch bedeutend reduzieren, da keine Bandlaufwerke oder optische Laufwerke mehr benötigt werden, die gewartet und erneuert werden müssen. Unternehmen zahlen dann einen monatlichen Betrag und Speichern ihre Daten den Firmennetz eines „Internet-Speicher-Unternehmens“.

Literaturverzeichnis

[ABC+02] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *5th Symposium on Operating Systems Design and Implementation (OSDI 2002), Boston, MA*, 2002.

[MMGC02] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benji Chen. Ivy: A read/write peer-to-peer file system. In *Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[Rab98] T. Rabin. A simplified approach to threshold an proactive rsa. In *InProceedings of Crypto*, 1998.

[RD01] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *In Proc. IFIP/ACM Middleware 2001, Heidelberg*, 2001.

[Sib03] Mike Sibler. Novel networking systems. In *Vortrag in Seminargruppe „Systeme für Hochleistungsrechnen“, Institute for Program Structures and Data Organization, Karlsruhe*, 2003.

[SMLN+02] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. In *To Appear in IEEE/ACM Transactions on Networking*, 2002.

[Tan95] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, Inc. Upper Saddle River, New Jersey, 1995.

[ZKJ01] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. In *Technical Report UCB//CSD-01-1141, U. C. Berkeley*, 2001.

Index

Äußerer Ring, 74
Überlagerungsnetzwerk
  semantisches, 1
Übertragungszeit, 5
'heart beat'-Anfrage, 79

Access Control List, 76
Anticipatory Scheduling, 8
asynchrone Aufträge, 11
Autentifizierung, 80
Autorisierung, 80

Bündel
  Rechner-, 1
Berechtigung, 80
Byzantine-Protokoll, 73

C-SCAN, 7
Caching, 78
  Read-Only-Dateien, 78
Caching, klassisch, 78
Caching, Log, 78
Caching, Schichten, 78
Chord, 72
Cluster, 1
Compress&Join-Algorithmus, 30
computer
  high-performance, 1
Content Hash Block, 72
Convergent Encryption, 81

Datei, 75
DHash, 72
Disk Scheduling, 1
Disk Scheduling, 3
distributed memory, 1

Effizienz, 29
Erasure Codes, 76
Exclusive Encryption, 81

Faden, 1
Farsite, 73, 76–81
Fehlertoleranz, 1
Festplatte, 4
FIFO, 6
File Diversion, 76
Freeblock-Scheduling, 14
FSCAN, 7

Gang Scheduling, 1, 22
Gitter
  Rechner-, 1
gleichberechtigte Rechnerknoten, 2

high-performance computer, 1
Hochleistungsrechner, 1

Innerer Ring, 74
Internet Wide Storage, 71
Ivy, 72, 76, 78–80

kontinuierlicher Algorithmus, 24
Koordinierung, 1

LAN, 1
LBN, 4
LIFO, 6
local area network, 1
Log, 76–78
lokaler Plattenspeicher, 1
lokales Netz, 1

Matrix-Algorithmus, 23
memory
  distributed, 1
memory access
  non-uniform, 1
message passing, 21

N-Step-SCAN, 7
network
  local area, 1
  semantic overlay, 1
  wide area, 1
Network Systems
  Novel, 2
Netz
  lokales, 1
Netzwerk
  Peer-To-Peer-, 2
  semantisches Überlagerungs-, 1
Netzwerkverbindungen, 1
Netzwerkweiter Speicherverbund, 2
non-uniform memory access, 1
Novel Network Systems, 2
NUMA, 1

Overlay Network
  semantic, 1
Overlay-Techniken, 2

Past, 72, 76–81
Pastry, 76

Peer-To-Peer-Netzwerk, 2
Performance-Driven Gang Scheduling, PDGS, 29
Plattenspeicher
  lokal, 1
Pond, 74, 76–81
Prioritätenverfahren, 6
Proactive Threshold Signatures, 81
Proportional-Share-Verfahren, 9
Prozess, 1
Prozessor, 1
Public-Key Block, 72

Rechner
  Hochleistungs-, 1
Rechnerbündel, 1
Rechnergitter, 1
Rechnerknoten
  gleichberechtigte, 2
Relikation, 79
Replica Diversion, 76, 79
Rotationsverzögerung, 5

SCAN, 7
Scheduling
  Disk, 1
  Gang, 1
Sektoren, 4
Selbstorganisation, 1
semantic Overlay Network, 1
semantisches Überlagerungsnetzwerk, 1
Sicherheit, 1, 79
space sharing, 22
Speedup, 29
Speicher
  verteilter, 1
Speichern von Dateien, 75
Speicherverbund
  netzwerkweiter, 2
Spur, 4
SSTF, 6
Suchzeit, 5

Tapestry, 75
thread, 1
time sharing, 22
Track-Aligned-Extents, 11

unzerteilter Algorithmus, 24

Verbund
  netzwerkweiter Speicher-, 2
Verschlüsselung, 79
Verschlüsselung, asymmetrisch, 80
Verschlüsselung, symmetrisch, 79
verteilter Speicher, 1
View, 72

WAN, 1
wide area network, 1

Zugriffszeit, 1, 5