Prüfer:	Prof. Dr. K. Lagally
Betreuer:	Dr. T. Schöbel-Theuer

Beginn am:	28.06.1999
Beendet am:	28.12.1999
CR-Nummer:	D.4.2 E.5

Diplomarbeit Nr. 1794

Entwicklung und Implementierung eines Backup- und Archivierungs-sytems mit Medienverwaltung

Jörg Matysiak
Matrikelnummer: 1667219

Institut für Informatik
Universität Stuttgart
Breitwiesenstrasse 20-22
D-70565 Stuttgart

Zusammenfassung

Diese Arbeit beschreibt die Entwicklung und Implementierung eines Backup- und Archivierungssystems.

Das System hat die Aufgabe eine hierarchische Speicherverwaltung zu realisieren:

Dateien, die in bestimmten Verzeichnissen abgelegt werden, sollen vom System automatisch auf billigere Speichermedien wie z.B. Magnetbänder ausgelagert werden. Für den Benutzer bleiben die ausgelagerten Dateien weiterhin sichtbar. Bei einem Zugriff auf eine ausgelagerte Datei wird diese vom System automatisch wiederhergestellt. Der zugreifende Prozeß wird bis zur erfolgten Wiederherstellung blockiert.

Im ersten Kapitel wird die Aufgabe des zu entwickelnden Systems beschrieben und ein kurzer Überblick über bereits existierende Anwendungen in diesem Bereich gegeben. Weiterhin werden die existierenden Anwendungen untereinander und mit dem zu erstellenden System verglichen.

Das zweite Kapitel behandelt die grundlegenden Teilproblematiken, die gefundenen Lösungsansätze und die mit deren Realisierung verbundenen Probleme.

Kapitel 3 beschreibt das Design des Programms und der notwendigen Schnittstellen.

Die Phase der Implementierung ist Thema des vierten Kapitels. Es werden die Aufgaben der einzelnen Teilprogramme und deren Module erklärt.

Das fünfte Kapitel zeigt nach einer Bewertung der gefundenen Lösung Überlegungen zu möglichen Erweiterungen und Änderungen des implementierten Systems auf.

Abstract

This document describes the development of a backup- and archiving-system.

The system's job is to realize a hierarchical storage managment:

Files that are placed in special directories should automatically get migrated to cheaper media like magnetic tapes. The files are still visible for users. If a user accesses a migrated file, the system automatically starts a recovering process. The accessing process is blocked until the file is recovered.

The first chapter explains the tasks the system should do and gives an overview over similar existing systems. It compares those existing systems with each other and with the system to be developed.

The second chapter takes a look at the basic problems, some solution ideas and problems while realizing that ideas.

Chapter 3 describes the design of the application and the needed interfaces.

The phase of implementation is the subject of the fourth chapter. It explains the modules and processes, their jobs and ideas.

In chapter 5 you can find an injudgement for the found solution and ideas for possible extensions and changes on the archiving-system.

COPYING

You can get a complete version of this document from

ftp://ftp.informatik.uni-stuttgart.de/fasy

Inhaltsverzeichnis

Abbildungsverzeichnis

Glossar

Die folgenden Anglizismen werden im Text verwendet. Für einige dieser Begriffe gibt es zwar auch deutsche Bezeichnungen, der Verständlichkeit halber werden aber die gebräuchlicheren, aus dem Englischen stammenden Begriffe benutzt.

Dateihandle: Ein Zeiger auf eine Datei. Ein Programm, das eine Datei öffnet, bekommt vom Betriebssystem einen Zeiger auf diese Datei, das **Dateihandle**. Das **Dateihandle** muß bei Zugriffen auf die Datei angegeben werden.

Flag: Zugriffsberechtigungsbit einer Datei.

Label: Kennung eines Mediums. **Labeln** eines Mediums bedeutet, ein Medium zu Initialisieren und mit einem **Label** zu versehen.

Inode: Ursprünglich I-Node (Index-Node). Eine kleine Tabelle, welche die Attribute und Plattenadressen der zu einer Datei gehörenden Blöcke verwaltet. Ein Inode wird selbst in einem Block auf der Festplatte gesichert.

Hardlink: Zuordnung eines Dateinamens zu einem Inode und damit zu einer Datei. Ein Inode kann mehrere Verzeichniseinträge innerhalb einer Festplattenpartition besitzen und daher unter verschiedenen Namen bekannt sein.

Sparsedatei: Eine Datei, die scheinbar eine bestimmte Größe besitzt, in Wirklichkeit aber nur ein einziges Zeichen enthält und auch nur einen Inode auf der Platte verbraucht. Eine Datei "zu **sparsen**" bedeutet, diese Datei durch eine gleichnamige **Sparsedatei** mit gleicher Größeninformation zu ersetzen.

Kernel: Kern eines Betriebssystems.

mounten: Montieren eines Dateisystems in die Verzeichnisstruktur. Das Abmontieren eines **gemounteten** Dateisystem wird auch **unmounten** genannt.

Kapitel 1

Einleitung

Die Einsatzgebiete von Computern sind vielseitig und fast täglich kommen neue hinzu. Dabei übernehmen sie viele Aufgaben die bisher von speziellen Geräten übernommen wurden.

So wird z.B. Musik im Internet vertrieben und auf Computern abgespielt, Faxe werden papierlos erstellt und empfangen, Computer ersetzen Anrufbeantworter, bearbeiten digitalisierte Fotos und Filme usw., die Liste ließe sich weiter fortsetzten.

Häufig treten bei solchen multi-medialen Anwendungen sehr große Datenmengen auf, die je nach Anwendungsgebiet für längere Zeit verfügbar sein müssen.

Ziel vieler Firmen ist das "papierlose Büro". Dabei werden beispielsweise Faxe nur digital empfangen und versendet. Diese geschäftliche Korrespondenz ist natürlich auch später noch für die Unternehmen wichtig, weshalb die erstellten oder eingegangenen Faxe nicht einfach gelöscht werden können.

Aus verschiedenen Gründen ist es nicht sinnvoll diese Daten auf einer lokalen Festplatte oder der Festplatte eines Servers vorzuhalten. Zum Einen ist die Kapazität von Festplatten beschränkt und die Speicherung auf Festplatten relativ teuer, zum Anderen besteht die Gefahr, bei einem Hardwaredefekt die Daten zu verlieren.

Eine andere Möglichkeit ist die Sicherung dieser Daten auf externen Medien wie z.B. Magnetbändern oder CD-ROM. Nachteil dieser Methode ist, daß hier der Benutzer in der Regel selbst wissen muß auf welchem Datenträger die Daten liegen, die er benötigt.

Da weder die eine, noch die andere Lösung wirklich befriedigend ist, werden z.B. Faxe heute sehr oft noch in ausgedruckter Form abgelegt. Diese Methode hat nicht zuletzt den Nachteil, daß eine digitale Weiterverarbeitung weitgehend ausgeschlossen ist.

Das in dieser Arbeit entstandene Programm versucht die Vorteile beider digitalen Archivierungsmethoden (Sicherung auf Festplatte und Sicherung auf externen Medien) zu verbinden. Die Archivierung von Daten und der spätere Zugriff darauf soll sich für den Benutzer wie der Zugriff auf eine Festplatte darstellen, deren Benutzung er ohnehin gewohnt ist. Die eigentliche Sicherung der Daten soll aber auf externen Medien stattfinden, so daß der Verbrauch an Festplattenkapazität in einem sinnvollen Rahmen liegt.

1.1 Aufgabenstellung

Die Aufgabe dieser Diplomarbeit ist die Entwicklung und Implementierung eines Backup- und Archivierungssystems, das große Datenmengen auf unterschiedlichen Medien (Magnetbändern, CD-R etc.) sichert.

Die Anwendung des Systems soll für den Benutzer möglichst einfach gestaltet werden - im Idealfall legt er die zu sichernden Dateien in einem bestimmten Verzeichnis (dem Client-Verzeichnis) ab und kann (abgesehen von einer zeitlichen Verzögerung) jederzeit lesend darauf zugreifen[1].

Auf einer Holding-Disk[2] existiert für jedes vom System benutztes Medium ein Spiegel-Verzeichnis, in welches die zu archivierenden Dateien kopiert werden. Erst in einem festgelegten Sicherungszeitraum (z.B. nachts) werden die Dateien von der Holding-Disk tatsächlich auf die Medien übertragen.

Die Dateien im Benutzerverzeichnis werden durch sogenannte Sparsedateien ersetzt, die "äußerlich" zwar die Größe der Originaldatei besitzen, real aber außer einem Inode keinen Speicherplatz auf der Platte belegen.

Auch die Dateien in den Medien-Verzeichnissen auf der Holding-Disk werden in Sparsedateien umgewandelt, sobald sie auf einem Medium gesichert wurden.

Durch die Konfiguration von Serien wird den Benutzern die Möglichkeit gegeben ihre Dateien zu kategorisieren und jede Kategorie auf unterschiedlichen Medien sichern zu lassen.

Wichtige Dateien können auch auf mehreren Serien und damit auf mehreren Medien gesichert werden, wodurch eine zusätzliche Sicherheitsstufe erreicht werden kann.

Die Handhabung der Speichermedien wird für den Benutzer weitestgehend transparent realisiert. Eingelegte Medien werden automatisch erkannt, leere Medien werden je nach Notwendigkeit initialisiert und einer bestimmten Serie zugeordnet.

Die Kommunikation mit dem Systemadministrator erfolgt per E-Mail. (Z.B. wird dem Administrator auf diese Art mitgeteilt welche Medien einzulegen sind.)

Die Archivierungsfunktionalität wird in einem eigenen Prozeß angesiedelt, mit dem über die Attribute und Zugriffszeiten von Dateien kommuniziert werden kann. Die notwendigen Verwaltungsinformationen müssen für den Benutzer nicht vollständig unsichtbar abgelegt werden, es ist durchaus denkbar, daß z.B. Dateien mit bestimmten Dateiendungen für Systemdaten reserviert sind.

Eine Wiederherstellung von Dateien kann durch Setzten bestimmter Zugriffsberechtigungen erreicht werden. Wird über einen modifizierten NFS-Dämon auf eine archivierte Datei zugegriffen, wird die Wiederherstellung automatisch, für den Benutzer transparent, angestoßen. Der Wiederherstellungsprozeß wird gestartet, sobald ein Benutzer versucht, auf eine Datei zuzugreifen (z.B. indem er versucht sie zu öffnen).

Als Sicherungsgeräte sind mehrere unterschiedliche Laufwerke denkbar; deshalb sollte der Zugriff auf die Geräte über Treiberprogramme erfolgen. Da das Programm auf `linde` [3] verwendet werden soll, ist ein Treiber für den dort eingebauten SONY-Bandroboter zu entwickeln.

[1]Ein nachträgliches Verändern von einmal archivierten Dateien ist nicht vorgesehen.

[2]Unter **Holding-Disk** versteht man einen Festplattenbereich, der zur Zwischenspeicherung von Daten reserviert ist, die auf einem Medium gesichert werden sollen.

[3]`linde` ist ein Rechner der Abteilung Betriebssoftware des Instituts für Informatik

Als Grundlage der Arbeit sollten möglichst externe Komponenten verwendet werden, die der BSD- oder GPL-Lizenz (Open Source) unterliegen. Der Code soll vorzugsweise in Perl, Java oder C mit `GNU autoconf` portabel unter Linux entwickelt werden. Das Ergebnis der Arbeit wird unter GPL veröffentlicht.

1.2 Softwarepakete mit ähnlichen Aufgaben

Daß Computer mit großen Datenmenge arbeiten ist nichts Neues, darum ist auch das Problem der Sicherung und Verwaltung großer Datenmengen nicht neu. Dieses Kapitel gibt einen kurzen Überblick über Eigenschaften und Besonderheiten verfügbarer Software zur Sicherung von Daten.

1.2.1 TAR/GNU TAR, CPIO, ZIP, RAR usw.

Es existieren zahlreiche Kommandozeilenprogramme, mit denen Dateien und Verzeichnisstrukturen in sogenannten Archiven zusammengefaßt werden können. Je nach Betriebssystem haben sich verschiedene Formate als Quasi-Standards durchgesetzt. (Z.B. werden unter Windows häufig `pkzip` und `rar` verwendet, unter Unix öfter `tar`).

Einige dieser Programme komprimieren die Archive gleichzeitig (z.B. `zip` und `rar`), andere erstellen unkomprimierte Archive (z.B. `tar`). Oft wird die Archivierung und Wiederherstellung der Dateien von ein und demselben Programm erledigt (z.B. `tar`, `cpio`), manchmal sind zwei verschiedene Programme dafür nötig (z.B. `zip`/`unzip`, `pkzip`/`pkunzip`). Einige Programm können selbstextrahierende Archive erzeugen, die allerdings nur betriebssystemspezifisch funktionieren (z.B. `pkzip`).

Die UNIX-Programme (z.B. `tar` und `cpio`) können in der Regel die Archive auch direkt auf Magnetbändern erzeugen bzw. von ihnen lesen.

Verwendungsbeispiel

Als Beispiel soll hier das unter UNIX verbreitete Programm `tar` näher erläutert werden:

Mit dem Kommandozeilenprogramm `tar` können Dateien und ganze Verzeichnisstrukturen in einem Archiv zusammengefaßt werden. Diese Archive können z.B. auf einer Festplatte (als Datei) oder auf Magnetbändern abgelegt werden. Bei der Wiederherstellung, die ebenfalls über `tar` erfolgt, können einzelne Dateien oder das gesamte Archiv wiederhergestellt werden.

Ein Beispiel für die Archivierung des Verzeichnisses `/home/joerg` auf Magnetband:

```
root@lukas:/> tar -cvf /dev/nst0 /home/joerg
```

Das Verzeichnis kann auch von `tar` wiederhergestellt werden:

```
root@lukas:/home> tar -xvf /dev/nst0
```

Weitere Informationen zu `tar` sind z.B. in [Pee93], Kapitel 20.05ff und [Fri95], Kapitel 10 zu finden, `cpio` wird in [Pee93], Kapitel 20.09 und [Fri95], Kapitel 10 näher erläutert.

1.2.2 DUMP/RESTORE

Das BSD-Programm `dump` ist ein Programm zur Archivierung von ganzen Dateisystemen (Partitionen). Mit dem Programm `restore` können von `dump` erstellte Archive wiederhergestellt werden.

`dump` taucht in verschiedenen UNIX-Derivaten auch als `backup` (AIX) oder `ufsdump` (Solaris) auf, `restore` heißt unter Solaris `ufsrestore`.

Es besteht die Möglichkeit verschiedene Sicherungsstufen auszuwählen. Sicherungen können sowohl auf externen Medien (z.B. Magnetbändern) als auch in Dateien ausgeführt werden.

Zu jeder gesicherten Datei vermerkt sich `dump` in einer Statusdatei, wann sie mit welcher Sicherungsstufe gesichert wurde. So kann verhindert werden, daß schon gesicherte Dateien bei jeder Sicherung erneut gesichert werden. Große Datenmengen kann `dump` auf mehrere Medien verteilen, so daß die zu sichernden Daten keiner Größenbeschränkung unterliegen.

Mit `restore` können aus den mit `dump` erstellten Archiven einzelne Dateien, Verzeichnisse oder gesamte Dateisysteme wiederhergestellt werden.

Weiterführende Informationen zur Anwendung von `dump` und `restore` sind [Fri95], Kapitel 10 und [Nem89], Kapitel 18 zu entnehmen.

1.2.3 CVS - Concurrent Versions System

Mit `CVS` und den ähnlichen Systemen `SCCS` und `RCS` können Dateien in verschiedenen Version archiviert werden. Die archivierten Daten werden selbst wieder auf einer Platte abgelegt. Deshalb werden die System hier zwar der Vollständigkeit halber erwähnt, aber nicht näher behandelt. (Näheres zu `SCCS` und `RCS` kann [Pee93] Kapitel 21.12ff entnommen werden, eine Beschreibung von `CVS` steht z.B. in [Gru])

1.2.4 AMANDA - Advanced Maryland Automatic Network Disk Archiver

`AMANDA` ist ein frei verfügbares System zur Archivierung von Partitionen in einem Netzwerk. Das Programm unterscheidet zwischen zwei verschiedenen Rechnertypen:

tape server host: Der *tape server host* ist der Rechner, an dem das Bandlaufwerk angeschlossen ist, auf dem die Daten gesichert werden.

backup client host: Die *backup client hosts* sind diejenigen Rechner im Netzwerk, deren Partitionen gesichert werden. Ein *tape server host* kann gleichzeitig auch ein *backup client host* sein.

Die Archivierung wird per `cron` zu einem Zeitpunkt gestartet, zu dem die Systembelastung des *tape server hosts* sehr gering ist. Von den *backup client hosts* können einzelne Partitionen oder ganze Festplatten gesichert werden.

Es gibt ein Programm zur Wiederherstellung einzelner Dateien, ganzer Partitionen oder Festplatten.

Von `AMANDA` benutzte Bänder sind durch ein Label von fremden Bändern und untereinander unterscheidbar, so daß falsch eingelegte Bänder nicht überschrieben werden. Bänder müssen manuell vom Benutze initialisiert ("gelabelt") werden.

Weitere Kennzeichen von AMANDA:

- in C geschrieben, lauffähig auf vielen Betriebssystemen (z.B. AIX, BSD, HP-UX, Linux, SunOS, u.a.)
- Administratorbenachrichtigung per E-Mail

Weitere Informationen zu AMANDA sind [Tea98] und [dS] zu entnehmen

1.2.5 ADSM - ADSTAR Distributed Storage Manager

Das IBM-Produkt ADSM[4] ist ein recht komplexes Archivierungs- und Sicherungssystem für eine große Anzahl unterschiedlicher Plattformen. Auch ADSM unterscheidet zwischen *Clients*, als den Rechnern, deren Daten ausgelagert werden, und *Servern*, welche die Daten letztlich auf die Medien schreiben.

Die Kennzeichen von ADSM sind:

- Client-Server-Architektur
- transparente Verteilung der Dateien auf die Medien (der Anwender muß nicht wissen, auf welchem Medium welche Datei gesichert ist)
- Kommandozeilenprogramme zur Steuerung der Archivierung (auch zur Verwendung in Skripten geeignet)
- grafische Oberfläche für Systeme mit X-Server
- Migration (automatische Aus- und Einlagerung von selten benutzten Dateien) wird unterstützt
- Dateien einzelner Client-Rechner können vor dem Zugriff anderer Clients geschützt werden, ein Schutz auf Benutzerebene existiert nicht.
- Unterstützung heterogener Umgebungen
- Unterstützung vielfältiger Plattformen
  - mehr als 35 Plattformen für Clients
  - als Server werden zur Zeit u.a. Windows NT, AIX, HP-UX, SUN Solaris, AS/400, MVS-OS/390, WM/ESA, OS/2 unterstützt

Mehr zu ADSM ist im Internet unter [wwwa], [wwwb], [wwwc], [wwwd], [wwwe] und [wwwf] zu finden.

1.2.6 OpenView OmniStorage

Der Grundgedanke von Hewlett-Packards OpenView OmniStorage ist ein hierarchisches Speichermanagment. Daten werden von teuren und schnellen Speichermedien (z.B. Festplatten) auf langsamere, billigere Medien (z.B. Magnetbänder) ausgelagert (migriert).

Weitere Merkmale von OpenView OmniStorage sind:

- Client-Server-Architektur
- transparenter Zugriff auf bereits archivierte Daten (über Export per NFS sogar von fast allen Betriebssystemen aus)

[4] ADSM wird mittlerweile von der IBM-Tochter Tivoli als Tivoli Storage Manager vertrieben

- graphische Benutzerschnittstelle
- Unterstützung heterogener Umgebungen
- Unterstützung von WORM[5]-Disks
- Integration in andere OpenView-Produkte (z.B. OpenView OmniBack) möglich
- Integration in das HP-UX-Dateisystem JFS, darum ist das Server-Programm nur unter HP-UX lauffähig.

Weitere Informationen sind im Internet unter [wwwg] und [wwwh] zu bekommen.

1.2.7 ARCserveIT

Ein weiteres Speichermanagment-System gibt es von *Computer Associates*. `ARCserveIT` sticht durch Unterstützung von *hot online backup and restore* hervor. Dies ermöglicht eine Sicherung oder Wiederherstellung von Dateien laufender Programme.

Eine weitere Besonderheit ist die Möglichkeiten Daten in RAID-Formaten, verteilt auf mehreren Bändern, zu sichern. Damit können die Daten auch bei Defekten einzelner Bänder noch wiederhergestellt werden.

Auf [wwwi] werden folgende Eigenschaften erwähnt:

- automatisches Speicher- und Medienmanagement
- Daten Migration
- “hot online backup/restore”
- RAID-Unterstützung
- Administratorbenachrichtigung per E-Mail
- Java-basierte Administratoroberfläche
- Server-Systeme: UNIX, Windows NT
- Client-Systeme: Windows 3.x/95/98/NT, NetWare, Apple Macintosh, OS/2, diverse UNIX-Derivate

1.2.8 SAM-FS

`SAM-FS` der Firma LSC Inc. ist ein Dateisystem zur Datenarchivierung mit folgenden Merkmalen (siehe auch [wwwj]):

- eigenes high-performance UNIX-Dateisystem
- Zugriffstransparenz
- Datensicherung im TAR-Format, Daten können auch ohne SAM-FS wiederhergestellt werden
- Java-basierte Administratoroberfläche
- Daten können direkt vom Medium in den Arbeitsspeicher gelesen werden (ohne den Umweg über die Festplatte)

[5]WORM (write once read multiple): Medien, die nur einmal beschrieben werden können

1.2.9 Begründung der Notwendigkeit eines neuen Systems

Nachdem nun einige Systeme mit zum Teil sehr umfangreichen Möglichkeiten vorgestellt wurden, ist zu Untersuchen, welche Vorteile ein neues System bringen würde.

Die bisherigen Systeme lassen sich grob in vier Gruppen einteilen:

1. Die kommandozeilenorientierten Systeme aus den Abschnitten 1.2.1 - 1.2.3: Sie verlangen explizite Eingaben vom Benutzer, wenn eine Sicherung stattfinden soll. Eine automatische Wiederherstellung der Daten oder eine Verwaltung der Medien wird von diesen Anwendungen nicht unterstützt.
2. AMANDA: Das System arbeitet zwar für den Benutzer transparent, eine Wiederherstellung der Daten erfolgt aber nicht transparent. Hier sind das Wissen und die Berechtigungen eines Administrators notwendig.
3. Die Systeme ADSM, ARCServeIT und SAM-FS: Sie benötigen eigene Software auf der Client-Seite. Dies schränkt die Verwendungsmöglichkeiten ein, auch wenn wie z.B. bei ADSM Clients für sehr viele Plattformen zur Verfügung stehen. (Die Client-Systeme müssen zumindest den Plattenplatz und die Rechenlast für die Client-Software zur Verfügung stellen.) ADSM bietet zudem für den einzelnen Benutzer nicht die Möglichkeit, Dateien vor dem Zugriff anderer Benutzer, die am gleichen Client-Rechner arbeiten, zu schützen.
4. Das Produkt OpenView OmniStorage: Es erfordert keine gesonderte Installation auf den Client-Rechnern. Der Server kann das Dateisystem über NFS exportieren. Für Benutzer steht somit eine Vielzahl von Client-Plattformen zur Verfügung. Ein großer Nachteil dieses Systems ist, daß zur Integration systemspezifische Eigenschaften des HP-UX Dateisystem JFS verwendet werden und darum als Server-Plattform nur HP-UX in Frage kommt.

Von den automatischen Systemen, die ihre Arbeit für den Benutzer transparent erledigen, ist nur AMANDA frei verfügbar. Alle anderen Systeme sind kommerzielle Produkte deren Preise sich in Bereichen mehrerer tausend DM bewegen.

Das entstehende System versucht nun die Vorteile der einzelnen Systeme zu Verbinden und einige neue Ideen einzubringen:

- Auf der Client-Seite muß keine gesonderte Software installiert werden, ein vorhandener NFS-Client reicht aus.
- Das Server-Programm wird unter Linux so entwickelt, daß ein Portierung auf andere System einfach Möglich ist.
- Das Programm soll unabhängig von den Eigenschaften des verwendeten Dateisystems funktionieren. (Sofern das Dateisystem POSIX-kompatibel ist.)
- Die Eingriffe in das vorhandene System sollen möglichst gering sein, das Archivierungssystem soll als normales Benutzerprogramm laufen und keine systemspezifischen Eigenschaften verwenden.
- Jeder Benutzer soll die Zugriffsberechtigungen seiner Dateien selbst bestimmen können.
- Die Anwendung wird unter GPL veröffentlicht und ist somit auch als Quellcode frei verfügbar. (Jeder darf das System kostenlos nutzen und eigene Erweiterungen und Änderungen einbauen.)
- Durch eine einfache, gut dokumentierte Treiberschnittstelle ist eine schnelle Anpassung an unterschiedliche Sicherungs-Hardware möglich.

- Das Serienkonzept wird eingeführt: Benutzer können zusammengehörige Daten auf gleiche Medien sortieren.
- Dateien können mehrfach gesichert werden, bei der Wiederherstellung kann ausgewählt werden, welche Kopie restauriert werden soll.
- Ein Lokationskonzept wird verwendet: Laufwerke und Serien können einer Lokation zugeordnet werden, neue Medien für eine Serie werden nur in Laufwerken mit gleicher Lokation initialisiert.

Kapitel 2

Lösungsansätze/High Level Design

Dieses Kapitel beschreibt Überlegungen, die während der Phase des Grob-Entwurfs angestellt wurden und begründet Designentscheidungen, wenn mehrere Alternativen zur Auswahl standen.

Zunächst werden die aus der Aufgabenstellung (Kapitel 1.1) enstehenden Anforderungen und die unter UNIX üblichen Techniken erläutert. Anschließend folgen Definitionen zu den neuen Ideen der Serie und der Lokation. Auf Grundlage dieser Voraussetzungen wird dann eine grobe Struktur des Systems entworfen.

Desweiteren werden Ideen zur Lösung grundlegender Problematiken und zur möglichen Programmgestaltung diskutiert. Am Ende des Kapitels werden schließlich mögliche Programmiersprache und -techniken zur Realisierung des Systems erörtert und die zur Realisierung getroffene Auswahl begründet.

2.1 Anforderungen an das System

In der Aufgabenstellung tauchen implizit und explizit folgende Anforderungen auf:

- Die Verwaltung großer Datenmengen muß möglich sein.
- Es sollen unterschiedliche Medien und Sicherungsgeräte unterstützt werden.
- Es gibt eine Holding-Disk, auf der die zu sichernden und die wiederhergestellten Dateien nach Medien-ID sortiert abliegen.
- Nicht mehr lokal vorhandene Dateien werden durch Sparsedateien ersetzt und behalten dadurch scheinbar ihre Größe.
- Dateien können in verschiedene Serien sortiert werden, verschiedene Serien werden auf unterschiedlichen Medien gesichert.
- Die Zuordnung von Dateien zu Serien und die Serienkonfiguration kann von jedem Benutzer selbst bestimmt werden.
- Die Benutzer müssen nicht wissen, welche Datei auf welchem Medium gesichert ist. Diese Information wird im System gehalten.
- Bei Problemen wird der Operator per E-Mail benachrichtigt.

- Um einen transparenten Zugriff auf die Daten zu erhalten, soll ein angepaßter NFS-Dämon verwendet werden.
- Die Archivierung selbst wird von einem eigenen Prozeß bewerkstelligt.
- Medien werden durch ein Label kenntlich gemacht.
- Leere Medien werden möglichst automatisch initialisiert.
- Dateien, die nicht vollständig auf ein Medium passen, werden auf mehrere Medien verteilt.

2.1.1 Grundlegende Techniken

Neben den in der Aufgabenstellung angegebenen Anforderungen sind noch einige grundlegende Regeln und Techniken zu beachten, die eine leichte Integration des Programms in die existierende Systemumgebung erlauben. Da das Programm zwar portabel, aber primär für UNIX-Systeme, erstellt werden soll, werden im Folgenden die verwendeten Konzepte aus der UNIX-Umgebung vorgestellt.

Das Dämonen-Prinzip

Unter UNIX werden Programme, die ihre Aufgaben im Hintergrund erfüllen, wie in [Tan95], Kapitel 7.3.1 beschrieben, als sogenannte **Dämonen** realisiert. Dämonen sind Programme, die keine Benutzerschnittstelle besitzen und die keine expliziten Eingaben verlangen. Es erfolgt keine Bildschirmausgabe, es existiert keine Möglichkeiten, Daten interaktiv einzugeben. In der Regel werden solche Programme beim Hochfahren des Systems automatisch gestartet, so daß der Benutzer von deren Existenz nichts bemerkt.

Da Dämonen keine Möglichkeit besitzen, Fehler interaktiv mitzuteilen, führen sie oft eine Protokolldatei (auch Trace- oder Logdatei), in der Ereignisse und Fehlermeldungen abgelegt werden.

Das Benutzerkonzept

Das UNIX-Benutzerkonzept schützt Anwender davor, daß Dateien absichtlich oder unabsichtlich durch unbefugte Benutzer oder Programme gelesen, beschädigt, gelöscht oder verändert werden. Einzig der Benutzer `root` hat uneingeschränkten Zugriff auf alle Dateien (vgl. [Kof98], Kapitel 5.1), sein Paßwort ist deshalb üblicherweise nur den Systemadministratoren bekannt.

Um ein System vor Schäden durch eine fehlerhafte Anwendung zu schützen, werden auch Dämonen oft nicht als `root`, sondern unter einem eigenen, eigens für dieses Programm eingerichteten Benutzer gestartet[1].

Das Konfigurationskonzept

Programme, die auf möglichst vielen Systemen lauffähig sein sollen, müssen eine bestimmte Anpassungsfähigkeit bieten. Eine Anpassung direkt im Code des Programms ist aus verschiedenen Gründen unakzeptabel. (Z.B. deswegen, weil der Benutzer zur Anpassung die verwendete Programmiersprache beherrschen muß.)

[1] beispielsweise wird unter *SuSE-Linux 6.0* der HTTP-Server *Apache* unter dem Benutzer `wwwrun` gestartet

Aus diesem Grund wird eine Anpassung meist über Konfigurationsdateien vorgenommen, die in einem festen Verzeichnis (z.B. `/etc`) abgelegt sind oder deren Pfad als Aufrufparameter übergeben werden kann.

Konfigurationsdateien sollen eine möglichst einfache und durchsichtige Anpassung ermöglichen und sind daher meist im Klartext verfaßt.

2.1.2 Definition von Serienkonzept und Lokationskonzept

Unter den verwirklichten Konzepten sind die Ideen der Serie und der Lokation neu. Deshalb ist hier zunächst eine Definition nötig.

Das Serienkonzept

Definition:

> Eine **Serie** bezeichnet eine Menge von Medien mit aufeinanderfolgenden Kennungen. Jede Serie besitzt eine Kennung, die Serienkennung. Diese Serienkennung ist Teil der Medienkennung eines jeden Mediums dieser Serie.

Serien können vom Benutzer oder vom Systemadministrator angelegt werden. Um zu verhindern, daß zwei Benutzer die gleiche Serie anlegen, hat jede Serie einen Besitzer, dessen Benutzerkennung Teil der Serienkennung ist. Als Serienbesitzer wird derjenige Benutzer definiert, dem die Datei gehört, in welcher die Serie konfiguriert ist.

Welche Datei in welche Serie gesichert wird ist vom Benutzer in den sogenannten Assoziationsdateien festzulegen.

Das Lokationskonzept

Definition:

> Eine **Lokation** bezeichnet einen Ort, dem eine Serie oder ein Gerät räumlich zugeordnet ist. Eine Serie oder ein Gerät kann gleichzeitig mehrere Lokationen besitzen.

Der Grundgedanke dieses Konzepts ist die Idee, daß ein Archivierungssystem eventuell Dateien auf verschiedenen Geräten in einem größeren Netzwerk sichern kann, bestimmte Daten aber z.B. aus Sicherheitsgründen nur lokal gesichert werden sollen.

Mit der Zuweisung einer oder mehrerer Lokationen zu einer Serie wird festgelegt: “Diese Serie soll nur auf Geräten mit einer dieser Lokationen gesichert werden.”.

Eine Prüfung der Lokation ist dabei nur bei der Initialisierung der Medien notwendig. So lange ein Medium an einem Ort verbleibt, ist damit gesichert, daß die Daten immer an diesem Ort gesichert und wiederhergestellt werden.

2.1.3 Weg einer Datei vom Benutzer zum Sicherungsmedium und zurück

In den beschriebenen Anforderungen ist schon der grundsätzliche Ablauf festgelegt, den eine Datei im zu entwickelnden System zu durchlaufen hat. Dateien die

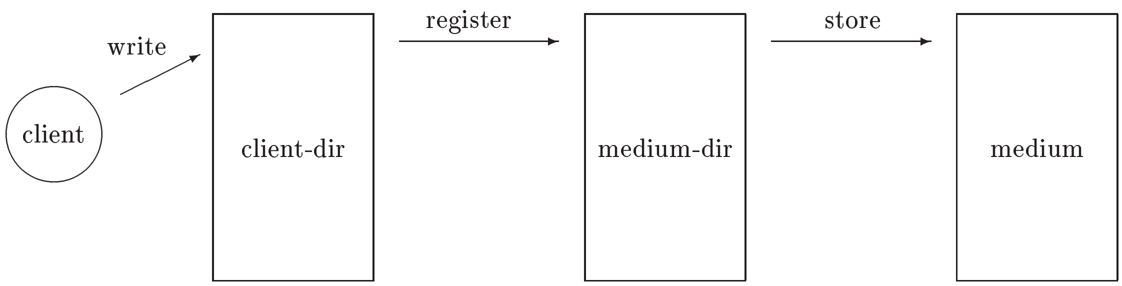


Abbildung 2.1: Vorläufiges Modell des Wegs einer Datei bei der Archivierung

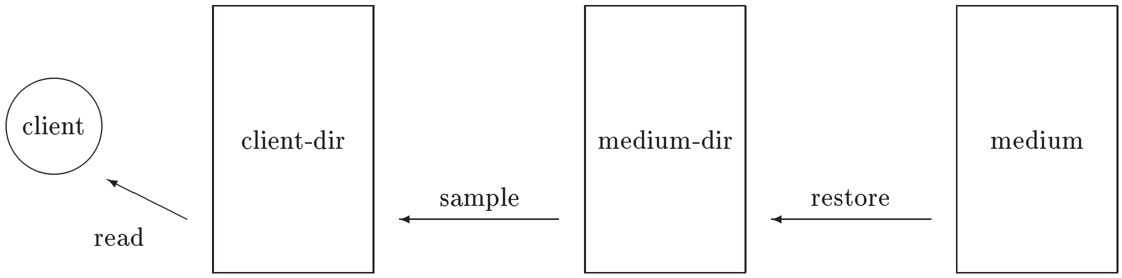


Abbildung 2.2: Vorläufiges Modell des Wegs einer Datei bei der Restauration.

vom System archiviert werden sollen, sind zunächst in einem Medien-Verzeichnis (medium-dir) der Holding-Disk zu sammeln. Dabei befinden sich in jedem Medien-Verzeichnis nur Dateien, die auch tatsächlich auf diesem Medium archiviert werden sollen. Paßt eine Datei nicht vollständig auf ein Medium, weil z.B. nicht mehr genügend Platz frei ist, wird sie auf mehrere Medien (der gleichen Serie) verteilt. Eine Sicherung der Dateien auf die Medien findet erst vom Medien-Verzeichnis aus statt. Der beschriebene Ablauf wird in Abbildung 2.1 verdeutlicht.

Umgekehrt werden zu restaurierende Dateien (bzw. deren Teile) zunächst vom Medium ins Medien-Verzeichnis kopiert und von dort ins Client-Verzeichnis übertragen (Und dabei ggf. zusammengesetzt, wenn eine Datei auf mehrere Medien aufgeteilt ist.) Abbildung 2.2 zeigt den Weg einer zu restaurierenden Datei.

Die einzelnen Schritte werden wie folgt definiert[2]:

Client-Verzeichnis (client-dir): Bezeichnet das Verzeichnis, in welches ein Benutzer bzw. das von ihm verwendete Programm Daten ablegt und über welches wieder auf die Daten zugegriffen werden kann.

Medien-Verzeichnis (medium-dir): Ist ein Verzeichnis, welches ein Medium repräsentiert. Es liegt auf der Festplatte und dient als "Zwischenlager" für Daten, die auf das Medium gesichert werden sollen bzw. Daten, die vom Medium

[2] in Klammern die in Abbildungen und im Programm verwendeten englischen Begriffe

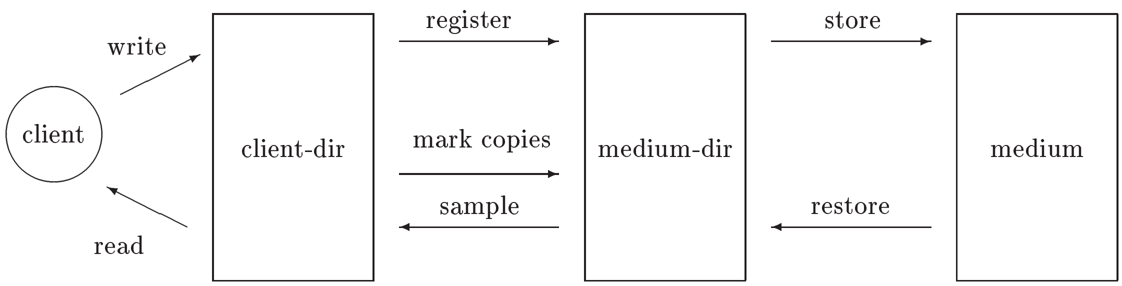


Abbildung 2.3: Endgültiges Modell sämtlicher Wege einer Datei

wiederhergestellt wurden.

Schreibzugriff eines Benutzers (write): Eine Datei wird im Client-Verzeichnis abgelegt.

Lesezugriff des Benutzers (read): Der Benutzer möchte lesend auf eine Datei zugreifen, z.B. indem er sie mit einem Programm öffnet.

Registrieren von Dateien (register): Neu abgelegte Dateien im Client-Verzeichnis werden als zu archivierend erkannt. Es wird eine Zuordnung zu Medien hergestellt und die Datei bzw. deren Teilkopien werden in die entsprechenden Medien-Verzeichnisse kopiert.

Sichern von Dateien (store): Dateien im Medien-Verzeichnis werden auf das zugehörige Medium übertragen.

Wiederherstellen von Dateien (restore): Dateien, die schon archiviert sind, werden vom betreffenden Medium gelesen und in das zugehörige Medien-Verzeichnis kopiert.

Zusammensetzen einer Datei (sample): Dateien, deren Teilkopien in den Medien-Verzeichnissen verfügbar sind (wiederhergestellt oder noch nicht gesichert), werden zur Originaldatei im Client-Verzeichnis zusammengesetzt. (Ein häufig auftretender Sonderfall ist der Fall, daß eine Datei nur aus einer einzigen Datei im Medien-Verzeichnis besteht.)

Der gesamte Archivierungsprozeß, der mit dem Schreiben der Datei durch den Benutzer beginnt und mit der Sicherung der Daten auf den Medien endet, wird im Folgenden als **Archivierung** bezeichnet. Der umgekehrte Prozeß, also die Wiederherstellung einer Datei aufgrund eines Lesezugriffs bis zu deren Zusammensetzung zur Originaldatei, wird von nun an **Restauration** genannt.

Im bisherigen Modell soll die Restauration durch einen Lesezugriff des Benutzers im Client-Verzeichnis ausgelöst werden. Der nächste Schritt, die Wiederherstellung der Dateien vom Band, hat allerdings keine direkte Verbindung zum Client-Verzeichnis, so daß unbekannt ist, welche Dateien wiederhergestellt werden müssen. Deshalb ist hier eine weitere Aktion nötig, welche die wiederherzustellenden Dateien in den Medien-Verzeichnissen markiert und so die Lücke schließt. Dieser neue Schritt wird als **markieren der Teilkopien (mark copies)** bezeichnet.

Damit ergibt sich nun ein vollständiges Bild des Ablaufs, wie es in Abbildung 2.3 zu sehen ist. Zu Beachten ist, daß auch "Abkürzungen" möglich sind. So kann der

Benutzer z.B. die Restauration einer Datei verlangen, die zwar registriert, aber noch nicht gesichert ist. In diesem Fall werden lediglich die in den Medien-Verzeichnissen noch vorhanden Teilkopien zusammengesetzt. Der rechte Teil der Grafik wird in diesem Fall gar nicht benutzt. (Die Teilkopien müssen natürlich trotzdem irgendwann gesichert werden, aber eine erfolgte Sicherung ist zur Restauration nicht zwingend notwendig.)

2.1.4 Initialisierung von Medien

Ein System, das Daten auf Medien sichern und wiederherstellen soll, muß eine Möglichkeit haben, die benutzten Medien voneinander zu unterscheiden. Dabei soll das System nicht nur seine eigenen Medien unterscheiden können, sondern auch Medien erkennen, die von anderen Anwendungen erzeugt wurden.

Eine verbreitete Methode, um dies zu erreichen, ist das Labeln von Medien. Dabei wird an einer festgelegten Stelle (meist am Anfang des Mediums) eine Kennung auf dem Medium abgelegt. Diese Kennung enthält neben der anwendungsspezifischen Bezeichnung des Mediums eine Identifizierung der Anwendung, die dieses Medium benutzt.

Die Aufgabenstellung verlangt, daß leere Medien vom System automatisch erkannt und mit einem Label versehen werden. Wann und wie dies geschehen soll, wird im Folgenden überlegt.

Leere Medien bedeuten freien Speicherplatz, der in der Regel knapp ist. Die Initialisierung (also das Labeln) eines Mediums weist dieses Medium fest einer Serie zu. Der Speicherplatz kann nun nicht mehr flexibel belegt werden, sondern nur noch von Dateien, die dieser Serie zugeordnet sind. Um möglichst lange flexibel zu bleiben sollte demnach versucht werden leere Medien erst zum spätmöglichsten Zeitpunkt zu labeln.

Dies führt zu dem Schluß, leere Medien erst dann zu initialisieren, wenn der Platz einer Serie zur Registrierung einer vom Benutzer abgelegten Datei nicht mehr ausreicht.

2.2 Das Treiberkonzept

Das entstehende Archivierungssystem soll möglichst flexibel sein und daher eine Anpassung an verschiedene Gerätetypen erlauben. Darum ist es nötig, für den Zugriff auf die Sicherungsgeräte eine Abstraktion einzuführen, eine sogenannte Treiberschnittstelle.

Die Schnittstelle definiert einen bestimmten Satz an allgemeinen Funktionen, die die Sicherungsgeräte anbieten müssen. Aufgabe des Treibers ist es, diese Funktionalität auf die real im Gerät vorhandenen Funktionen abzubilden.

Neue Treiber sollen einfach und dynamisch eingebunden werden können, ohne Veränderungen am Rest des Systems vornehmen zu müssen. Ein Treiberkonzept auf Basis von Programm-Modulen beschränkt die Entwickler von Gerätetreibern auf eine einzige Programmiersprache. Um auch in diesen Punkt flexibel zu bleiben, wird folgendes Konzept verwendet:

- Jedem Gerät wird ein Gerätetyp zugewiesen.
- In einem Treiber-Verzeichnis gibt es für jeden Gerätetyp ein Unterverzeichnis.

- In jedem dieser Unterverzeichnisse existiert eine festgelegte Menge von Programmen, welche als Einheit den Treiber darstellen.
- Die Aufrufparameter und Verhaltensweisen der Programme sind dokumentiert.
- Um die Entwicklung von Treibern zu vereinfachen wird eine Bibliothek von allgemeinen Funktionen implementiert, die von den Treibern benutzt werden können. (Damit wird den Entwicklern zwar eine Programmiersprache nahegelegt, aber eine starre Festlegung ist nicht erfolgt, da die Entwicklung eines Treibers auch ohne diese Bibliothek möglich ist.)

2.3 Datenhaltung

Ein Archivierungssystem soll so konstruiert sein, daß ein Abbruch und Neustart des Programms, ob beabsichtigt oder unabsichtlich (z.B. durch einen Absturz), nicht zum Verlust von Daten führt.

In diesem Abschnitt wird überlegt, wie Daten möglichst persistent (über einen Programmneustart hinweg) und konsistent (widerspruchsfrei) gehalten werden können.

2.3.1 Konsistente und persistente Datenhaltung

Jedes Programm braucht zum Betrieb irgendwelche Daten, die in der Regel im Hauptspeicher abgelegt werden. Daten, die auch bei einem erneuten Start eines Programms verfügbar sein sollen (z.B. Einstellungen eines Editors), werden hingegen in Dateien gesichert.

Wie in der Aufgabenstellung schon angedeutet, bietet es sich an, den Zustand einer Datei in deren Zugriffsberechtigungs-Bits (Flags) zu sichern. Eine solche Sicherung ist in jedem Fall konsistent (ein Bit ist entweder gesetzt oder nicht) und persistent (selbst nach einen Neustart des Rechners behalten die Flags ihre Werte).

2.3.2 "Quasi-atomar" erzeugte Dateien

Komplexere Daten, die nicht durch ein einzelnes Bit dargestellt werden können, müssen in Dateien gesichert werden. Hier ergibt sich allerdings das Problem, daß evtl. inkonsistente Informationen auftreten können. Dies ist z.B. der Fall, wenn ein Programm während des Schreibens einer Datei beendet wird. Die Datei enthält dann eventuell unvollständige Informationen, was zu Fehlinterpretationen führen kann.

Gesucht wird eine Lösung, in der eine Datei nur in einem der beiden Zustände "vollständig vorhanden" oder "gar nicht vorhanden" vorkommen kann. Es ist also eine Möglichkeit zu finden, eine atomare Erzeugung der Dateien zu simulieren.

Hier hilft eine Eigenschaft des Dateisystems: die Umbenennung von Dateien erfolgt atomar, indem ein Eintrag im Verzeichnis geändert wird (Zuordnung *Dateiname* $\rightarrow$ *Inode*). Um eine Datei quasi-atomar zu erzeugen müssen die Daten also zunächst in eine temporäre Datei geschrieben werden. Sind die Daten vollständig geschrieben, wird diese Datei in ihren Originalnamen umbenannt und erscheint so als atomar erzeugt. [3]

Es muß allerdings verhindert werden, daß die temporären Dateien interpretiert werden. Dies kann über folgende Regeln sichergestellt werden

- Es gibt feste Regeln zur Erzeugung temporärer Dateinamen.
- Temporäre Dateinamen beginnen mit einem Punkt und sind somit unter UNIX meist unsichtbar. (Programme wie z.B. `ls` zeigen solche Dateien normalerweise nicht an).
- Beim Programmstart werden alle alten (von früheren Programmläufen übriggebliebenen) temporären Dateien entfernt.
- Die vom Programm verwendeten Dateinamen unterscheiden sich von allen durch diese Regeln erzeugbaren temporären Namen.

2.3.3 Kopien oder Hardlinks von Dateien erstellen

Das Konzept der Medien-Verzeichnisse könnte dazu führen, daß Dateien bis zur Sicherung mehrfach im System vorhanden sind: Einmal im Client-Verzeichnis und einmal in jedem Medienverzeichnis dem die Datei zugeordnet wurde. Durch diese Konstruktion verbraucht jede Datei ein mehrfaches ihrer Größe an Plattenplatz. Dies ist natürlich nicht gewünscht, denn der Sinn des Archivierungssystem ist das Einsparen von Speicherplatz.

Ein Großteil des benötigten zusätzlichen Speicherplatzes kann eingespart werden, indem statt Kopien Hardlinks dieser Datei angelegt werden. Wird von einer Client-Datei ein Hardlink erzeugt, so bedeutet dies, daß der Inode dieser Datei über einen zusätzlichen Verzeichniseintrag erreichbar ist. Ein Hardlink ist einfach ein zweiter Name für die gleiche Datei. Da die Datei nur einmal physikalisch vorhanden ist verbraucht sie den Speicherplatz nur einfach.

Ein weiterer Vorteil dieser Methode ist, daß das Erstellen von Kopien wesentlich schneller möglich ist. (Die Datei muß nicht kopiert werden, es wird nur ein Verzeichniseintrag angelegt.)

Diese Methode funktioniert allerdings nur mit vollständigen Kopien. Wird eine Datei in Teilen auf verschiedenen Medien gesichert, müssen nach wie vor echte Teilkopien erstellt werden.

Nachteilig ist, daß Hardlinks nicht über Dateisystemgrenzen hinweg angelegt werden können. Sie sind auf eine Partition beschränkt. Da das Programm vor einer Kopier-Aktion nicht wissen kann, ob zwei Verzeichnisse auf dem gleichen Dateisystem liegen, wird der in Abbildung 2.4 als Nassi-Schneiderman-Diagramm verdeutlichte Algorithmus zur Erstellung von Kopien verwendet.

[3] Diese Logik funktioniert allerdings nur in einem transaktionsbasierten Dateisystem wirklich sicher. In anderen Dateisystemen kann ein Stromausfall oder Systemabsturz dazu führen, daß Dateien nach einer Reparatur des Dateisystems (z.B. über `fdisk`) inkonsistent wiederhergestellt werden. Ein transaktionsorientiertes Dateisystem für Linux ist z.B. das von SGI entwickelte XFS, siehe [wwwn].

Kopier-Logik — Verwendung von Hardlinks

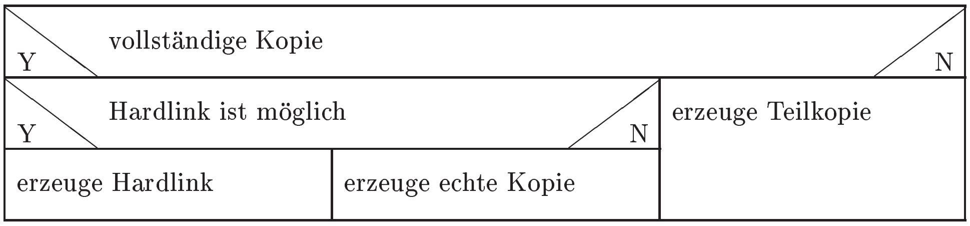


Abbildung 2.4: Nassi-Schneiderman-Diagramm der Kopier-Logik

2.4 Multi-Prozeß-Konzept

Ein Archivierungssystem ist keine zeitkritische Anwendung. Der Zugriff auf Daten die nur selten benötigt werden dauert meist schon wegen der schlechten Zugriffszeiten der Medien (z.B. Magnetbänder, CD-ROMs) lange. Deshalb erscheint es auf den ersten Blick wenig sinnvoll ein Archivierungssystem aus mehreren Prozessen aufzubauen und damit Probleme wie Synchronisation und Interprozeßkommunikation auf sich zu nehmen.

Warum die Entscheidung trotzdem zugunsten eines Mehrprozeßsystems gefallen ist und wie die angesprochenen Probleme gelöst werden wird im Folgenden erklärt.

2.4.1 Parallelisierung

Zugriffe auf Geräte dauern im Vergleich zu Speicherzugriffen relativ lange. Im Fall von Sicherungsgeräten, insbesondere Bandlaufwerken und -robotern, kann ein Zugriff mehrere Sekunden oder gar Minuten dauern. (Z.B. dauert das Laden eines Mediums aus einen Slot bei dem in `linde` eingebauten Bandroboter je nach Gerätezustand tatsächlich mehrere Minuten.)

Die Sicherung und Wiederherstellung von sehr großen Dateien wird sehr viel Zeit in Anspruch nehmen. In einem System mit mehreren Laufwerken ist es daher sinnvoll, Gerätezugriffe parallel ablaufen zu lassen. Dadurch wird der einzelne Zugriff zwar nicht schneller, die Geschwindigkeit des Systems insgesamt erhöht sich aber merklich.

Man stelle sich z.B. ein System mit 2 Bandrobotern vor, die jeweils 20 Sekunden zum Laden eines Bands benötigen. In einem sequentiellen System würde zunächst der erste Roboter das gewünschte Band laden und seine Operationen darauf ausführen, was beispielsweise weitere 50 Sekunden dauern würde. Erst nach dieser Zeit (20 + 50 = 70 Sekunden) beginnt der zweite Roboter mit dem Laden des Bands, was ebenfalls 20 Sekunden dauert. Zuzüglich weiterer 50 Sekunden zur Ausführung der Operation sind insgesamt 140 Sekunden für beide Operationen vergangen.

In einem parallelen System werden beide Roboter kurz nacheinander beginnen die Bänder zu laden und ihre Operationen darauf auszuführen. Nach ca. 70 Sekunden sind alle Operationen ausgeführt und beide Geräte wieder betriebsbereit.

Sollen sehr große Datenmengen (z.B.mehrere Gigabyte) auf oder von Medien übertragen werden, ist davon auszugehen, daß die Übertragung mehrere Minuten dauert. In einem System mit mehreren Sicherungsgeräten und mehreren Benutzern sollte vermieden werden, daß eine Sicherung bzw. Wiederherstellung großer Dateien den Rest des Systems blockiert. Eine solche Blockade kann nur durch eine Parallelisierung der Gerätezugriffe vermieden werden.

Wenn das System in mehrere parallel laufende Prozesse aufgeteilt wird, so ist zu befürchten, daß diese Prozesse sich gegenseitig stören könnten. Es ist also sorgfältig zu überlegen, bei welchen Aktionen Parallelität sinnvoll ist, und wo sie nur weitere Probleme aufwirft.

Als Funktionen bei denen eine parallele Bearbeitung Vorteile bringt, kommen Aktionen in Frage, die unter Umständen sehr lange dauern können und die weitgehend unabhängig vom Rest des Systems ausgeführt werden können.

Wie oben bereits erwähnt, können Gerätezugriffe eventuell sehr lange dauern. Die Zugriffe auf Geräte können bei geeigneter Synchronisation auch unabhängig vom Rest des Systems erfolgen. Zu Parallelisieren sind daher folgende Aktionen:

- Geräteprüfungen ("Welches Medium liegt in welchem Slot?")
- Schreiboperationen auf Medien
- Leseoperationen auf Medien

Alle anderen Aktionen greifen nur auf die Festplatte bzw. den Hauptspeicher zu und sind daher so schnell, daß eine Parallelisierung im beschriebenen System nicht notwendig ist.

2.4.2 Synchronisation

Die Synchronisation der parallelisierten Operationen beschränkt sich auf den Zugriff auf die von ihnen benutzten Geräte. Es muß sichergestellt werden, daß kein anderer Prozeß diese Geräte gleichzeitig benutzt.

Eine Synchronisation auf den Zugriff der Dateien in den Medienverzeichnissen ist nicht nötig, da diese über den in Abschnitt 2.3.2 beschriebenen Algorithmus erzeugt werden und somit entweder vollständig oder gar nicht vorhanden sind.

Eine einfache und wirksame Lösung des Synchronisationsproblems ist die Verwaltung aller Ressourcen durch einen zentralen Prozeß, der die Berechtigung zur Nutzung einer Ressource an andere Prozesse weitergibt. Dieser Prozeß wird im folgenden **Hauptprozeß** genannt.

Ein Teilproblem kann mit dieser Strategie allerdings nicht gelöst werden: Der Zugriff eines fremden, nicht zum Archivierungssystem gehörenden Programms auf vom System benutzte Geräte. Eine praktikable vollständige Lösung dieses Problems ist nicht bekannt. Allerdings können mögliche Fehlerquellen weitgehend ausgeschlossen werden, wenn die Gerätedateien der zu benutzenden Geräte dem Benutzer des Archivierungssystems gehören und nur dieser Berechtigungen auf dem Gerät besitzt. Somit können nur der Archivierungsbenutzer selbst und `root` auf das Gerät zugreifen. Die Kennwörter beider Benutzer sollten nur den Systemadministratoren zugänglich sein, welche wissen sollten, daß die Geräte nicht benutzt werden dürfen.

2.4.3 Interprozeßkommunikation

In einem System, das aus verschiedenen Prozessen besteht, muß es eine Möglichkeit zum Datenaustausch zwischen diesen Prozessen geben.

Im beschriebenen System sind zwei Richtungen der Datenübermittlung notwendig: Die Prozesse, denen eine Geräte-Nutzungs-Berechtigung zugeteilt wird müssen wissen, auf welches Gerät sie zugreifen dürfen. Umgekehrt müssen sie dem Hauptprozeß die Ergebnisse ihrer Arbeit und die Freigabe des Geräts mitteilen können.

Zuteilung einer Geräte-Nutzungs-Berechtigung

Wie oben beschrieben kommen als asynchrone (parallel laufende) Prozesse nur Gerätestatusprüfung und Schreib-/Leseoperationen in Frage. Diese Funktionen müssen wegen der Abhängigkeit von der Hardware ohnehin in Treiberprogrammen realisiert werden (denn Magnetbänder müssen bspw. anders beschrieben werden als CD-ROMs).

Um zusätzliche Komplexität zu vermeiden, bietet es sich an, die Parallelität zu erreichen, indem die Treiberprogramme asynchron, also losgelöst vom eigentlichen System, gestartet werden. Desweiteren kann festgelegt werden, daß der Hauptprozeß Gerätetreiber-Programme nur dann startet, wenn das Gerät, auf das zugegriffen werden soll, frei ist.

Ein gestarteter Treiberprozeß kann davon ausgehen, daß er ohne Einschränkungen auf ein Gerät zugreifen kann, sobald er gestartet ist. Erst wenn dieser Treiberprozeß beendet wurde geht die Kontrolle über das Gerät an den Hauptprozeß zurück.

Nun fehlt lediglich eine Möglichkeit, dem Prozeß mitzuteilen welches Gerät er benutzen darf und welche Aktionen er auszuführen hat. Für solche Zwecke werden in der Regel Kommandozeilenparameter gebraucht. Da alle notwendigen Daten schon vor dem Start eines Treiberprozesses feststehen, reicht diese Technik in diesem Fall vollkommen aus.

Bekanntmachen der Ausführungsergebnisse

Jeder gestartete Prozeß hat eine bestimmte Aufgabe auszuführen, deren Ergebnis im Gesamtsystem verwendet werden soll.

In den beschriebenen parallelisierten Prozessen sollen folgende Daten an den Hauptprozeß zurückgegeben werden:

Gerätestatusprüfung: Status des Geräts (z.B. Gerät ist benutzbar/nicht benutzbar), Kennungen der eingelegten Medien, etc.

Leseoperation: gelesene Daten, Erfolgs- oder Mißerfolgsmeldung

Schreiboperation: Erfolgs- oder Mißerfolgsmeldung

Es können also durchaus komplexere Ergebnisse anfallen, die an den Hauptprozeß übermittelt werden müssen.

Lösungen zur Interprozeßkommunikation gibt es sehr viele (z.B. Messagequeues, Shared Memory, FIFOs, etc). Leider ist deren Verfügbarkeit und Anwendung meist sehr stark vom verwendeten Betriebssystem abhängig (vgl. auch [Bec99]).

Eine einfache und auf praktisch allen Systemen verfügbare Lösung ist die Kommunikation über Dateien. Die dabei auftretenden Probleme wurden in Abschnitt 2.3.2 bereits gelöst.

Prozesse, die vom Hautprozeß gestartet wurden und Ergebnisse an diesen zurückgeben möchten, schreiben ihre Ergebnisse in eine Ergebnisdatei. Diese Ergebnisdatei wird für den Hauptprozeß erst sichtbar, wenn sie vollständig geschrieben wurde.

Die Ergebnisdateien werden in einem vom Hauptprozeß verwalteten Verzeichnis abgelegt. Hat der Hauptprozeß eine Ergebnisdatei eingelesen und interpretiert, löscht er sie. Um eine Eindeutigkeit in der Namensgebung der Ergebnisdateien zu erreichen, enthalten deren Namen die Kennung des Geräts, auf das der Prozeß zugegriffen hat. Da nur ein Prozeß gleichzeitig auf ein Gerät zugreifen kann wird auch nur dieser Prozeß eine Ergebnisdatei mit der Kennung des Geräts erzeugen. Somit ist gesichert, daß kein anderer Prozeß die Ergebnisdatei überschreibt.

2.5 Zugriffsberechtigungen

Um größere Auswirkungen bei Programmfehlern oder Fehlkonfigurationen zu vermeiden, soll das Programm unter einem eigenen Benutzer (**Archivierungsbenutzer** genannt) laufen.

Bei einem System, das die Dateien verschiedener Benutzer archivieren soll, ergeben sich dadurch verschiedene Probleme:

Die UNIX-Zugriffsberechtigungen schützen Dateien eines Benutzers vor dem Zugriff anderer Benutzer, also auch vor dem Zugriff des Archivierungsbenutzers. Der Archivierungsbenutzer benötigt aber den Zugriff auf die zu archivierenden Dateien, z.B. zum Lesen der Dateien und zum Setzten der Flags.

Eine Möglichkeit wäre das Programm als `root` auszuführen, was aber nach Möglichkeit vermieden werden soll. Eine andere Idee ist, nur solche Programmteile mit `root`-Berechtigung auszustatten, die diese auch wirklich benötigen.

Dazu müssen die betroffenen Funktionen als eigene Prozesse mit `root`-Berechtigung gestartet werden. Hierfür gibt es folgende Möglichkeiten:

1. Die Programme gehören dem Benutzer `root` und sind mit den sogenannten SUID- und SGID-Flags ausgestattet.[4]

2. Die Programme werden über das Programm `sudo` mit `root`-Rechten ausgestattet.[5]

Die erste Variante bietet den Vorteil, daß dies von den meisten UNIX-System bereits unterstützt wird. Nachteil dieser Methode ist, daß in einigen UNIX-Systemen (u.a. Linux, vgl. [Fen98], Abschnitt 5.2) die Ausführung von Skripten als SUID-Programme verboten ist.

Die Benutzung von `sudo` bringt die Vorteile einer genaueren Konfigurierbarkeit und die Möglichkeit, auch Skripte ausführen zu können. Nachteilig ist, daß ein zusätzliches Programm installiert werden muß, wenn es nicht bereits installiert ist.

Aus Gründen der Sicherheit (es kann genau festgelegt werden welcher Benutzer welches Programm unter welcher Benutzerkennung ausführen kann) und der Flexibilität wird die zweite Variante (`sudo`) verwendet.

[4] Die Datei-Flags SUID und SGID bewirken, wenn sie gesetzt sind, daß ein Programm unter der Benutzer- bzw. Gruppenkennung des Dateibesitzers gestartet wird. Normalerweise werden Programme unter der Kennung des Benutzers gestartet der das Programm aufruft.

[5] Das Programm `sudo` ermöglicht Benutzern Programme unter der Kennung eines anderen Benutzers auszuführen.

2.6 Konfiguration des Programms

Vor der Entwicklung des Programms ist genau zu überlegen welche Parameter anpaßbar sein müssen und bei welchen Parametern man von festen Größen ausgehen kann. Werden mögliche Veränderliche vergessen, kann dies während der Implementierungsphase dazu führen, daß das gesamte Konzept nocheinmal verändert werden muß und die bisherige Arbeit zu großen Teilen wertlos wird.

Im Gegensatz zu Anwendungen unter Windows oder OS/2, die Einstellungen oft in binären Formaten sichern, werden Einstellungen unter UNIX üblicherweise in lesbaren Klartext-Dateien gehalten.

Im vorliegenden System wird zwischen zwei verschiedenen Einstellungsmöglichkeiten unterschieden: Zum Einen gibt es die *allgemeinen Einstellungen*, die für das gesamte System gelten und deshalb vom Systemadministrator vorzunehmen sind und zum Anderen die *benutzerspezifischen Einstellungen*, die jeder Benutzer selbst verändern kann, sofern er dazu berechtigt ist.

In den kommenden Abschnitten werden die notwendigen Parameter und mögliche Einstellungen erklärt.

2.6.1 Allgemeine Einstellungen

Die allgemeinen Einstellungen beziehen sich auf system- und hardwarespezifische Eigenschaften, sowie auf das Verhalten des Programms insgesamt.

Konfigurierbar gestaltet werden:

- sämtliche Pfade zu Programmaufrufen oder Dateiablage, z.B. die Pfade zu Gerätetreibern, Client- und Medien-Verzeichnissen, temporären Verzeichnissen
- Zahl, Art und Eigenschaften der verwendeten Sicherungsgeräte, z.B. lesbare/beschreibbare Medientypen, Anzahl der Slots, Lokation
- Umgang mit Fehlermeldungen, Name und maximale Größe der Logdatei, Aufruf eines Programms zur Erzeugung einer E-Mail
- Einstellungen zu Datenträgern, z.B. Typ, Kapazität
- Zeiträume in denen eine Sicherung stattfinden soll
- Zeiträume, nach denen eine Datei registriert oder gesichert werden darf (um z.B. zu verhindern, daß eine noch nicht vollständig übertragene Datei registriert wird)
- Zeiträume, nach denen eine Datei vollständig aus dem System entfernt wird

2.6.2 Benutzerspezifische Einstellungen

Die Einstellungsmöglichkeiten der Benutzer beschränken sich zunächst auf die Zuordnung *Datei* $\rightarrow$ *Serie* und die Definition von Serien.

Um den Benutzern eine möglichst flexible Konfiguration zu ermöglichen, werden die benutzerspezifischen Einstellungen in Konfigurationsdateien in den Client-Verzeichnissen vorgenommen[6]. Dabei überschreiben Einstellungen in tieferen Verzeichnissen diejenigen von übergeordneten Verzeichnissen.

Da nicht jeder Benutzer die Berechtigung zur Veränderung des ein oder anderen Parameters haben soll, wird zusätzlich eine Möglichkeit zur Beschränkung der Benutzerrechte eingeführt. Hier gilt die Regel, daß Einstellungen übergeordneter Verzeichnisse in tieferliegenden nur eingeschränkt, nicht aber erweitert werden können. (Ein Benutzer der in einen übergeordneten Verzeichnis bspw. keine Serien anlegen darf, darf dies auch in keinem Unterverzeichnis tun.)

2.7 Programmiertechnik

Nachdem das grundsätzliche Programmverhalten und erste Teilproblematiken durchdacht wurden, muß nun entschieden werden, wie das Programm aufgebaut werden soll und nach welcher Technik es funktionieren soll.

In Kapitel 2.3 wurde entschieden, relevante Daten nicht im Hauptspeicher zu halten. Jegliche Information, die von Dauer sein soll, wird auf der Festplatte in Dateien oder deren Flags gehalten. Man kann die Flags der Datei somit als eine Zustandsbeschreibung interpretieren: Eine Datei durchläuft eine Menge von Zuständen. In welchem Zustand sie sich momentan befindet, kann an ihren Flags erkannt werden.

Das Archivierungssystem stellt gewissermaßen einen Automaten dar, der Dateien in einem Netz von Zuständen bewegt.

Das Programm läuft in einer Schleife, die mit der Untersuchung der Festplatte und der Laufwerke beginnt. In diesem Schritt wird festgestellt, welche Dateien sich in welchen Zuständen befinden und welche Medien eingelegt sind. Für jede Datei, deren Zustand eine Aktion verlangt, wird geprüft, ob die Aktion ausgeführt werden kann. Sind die Vorbedingungen einer Aktion erfüllt, wird diese ausgeführt.

Als Ergebnis einer Aktion befindet sich die Datei in einem anderen Zustand. Solange eine Aktion nicht erfolgreich ausgeführt wurde, verbleibt eine Datei in ihrem alten Zustand. Dies kann unter Umständen dazu führen, daß eine Aktion mehrfach ausgeführt wird, dies ist beim Design der Aktionen zu beachten.

Die Implementierung legt einen Satz von Regeln fest. Eine Regel besteht aus einer Bedingung und einer Aktion. Die Aktion wird nur ausgeführt, wenn die Bedingung erfüllt ist. Eine Bedingung zur Sicherung einer Datei kann verbal beschrieben z.B. so aussehen [7]:

```
Datei befindet sich im Zustand "zu sichern" +
das betreffende Medium ist eingelegt +
ein Gerät zum Schreiben des Mediums ist vorhanden
```

Die dazugehörige Aktion wäre dann z.B.:

```
Lade das Medium und sichere die Datei,
überführe die Datei in den Zustand "gesichert"
```

Diese Art der regelbasierten Programmierung hat den Vorteil, daß Eingriffe des Benutzers vom System toleriert werden. (Z.B. wenn der Benutzer Medien auswechselt oder manuell mit einem Label versieht.) Auch Programmabstürze oder -abbrüche

[6] Hier muß natürlich darauf geachtete werden, daß die Konfigurationsdateien nicht versehentlich archiviert werden

[7] "+" bedeutet eine UND-Verknüpfung

können ohne Probleme gehandhabt werden, denn jegliche Zustandsdaten des Systems werden persistent auf der Platte gesichert und sind somit nach einen neuen Programmstart weiterhin verfügbar.

Ein weiterer Vorteil dieser Technik ist, daß die Teilprogramme, in diesem Fall die Bedingungen und Aktionen, losgelöst vom Restprogramm implementiert und getestet werden können. Wichtig ist nur, daß das Design der Zustände und Zustandübergänge stimmt, so daß das Zusammenspiel der Regeln funktioniert.

2.8 Wahl der Programmiersprache

In der Aufgabenstellung wird die Menge der möglichen Programmiersprachen auf die drei mehr oder weniger portablen Sprachen *Perl*, *Java* und *C* beschränkt. Im Folgenden werden die drei Sprachen kurz im Hinblick auf die zu lösende Aufgabe und die in den vorangegangenden Kapiteln getroffenen Design-Entscheidungen verglichen.

Dabei werden vor allem die folgenden Merkmale untersucht:

- Portierbarkeit
- Möglichkeiten zur Erzeugung und Interpretation von Textdateien (Konfigurationsdateien, Ergebnisdateien zur Interprozeßkommunikation)
- Funktionen und Operationen zur Prüfung und Änderung von Dateiattributen
- Aufruf externer Programme/Skripte (z.B. für Treiber-Programme)
- Möglichkeiten zum Aufbau von Datenstrukturen
- Möglichkeiten zur automatischen Erstellung von Code-Dokumentation

Perl

Perl ist eine Skriptsprache mit C-ähnlicher Syntax und automatischer Speicherverwaltung (Garbage Collection). Perl ist auf sehr vielen Plattformen verfügbar (siehe [Wal99]). Programmierern läßt Perl sehr großen Freiraum im Hinblick auf Programmiertechniken, es werden z.B. objektorientierte Programmierung ebenso unterstützt wie Modul-Konzepte. Eines der Grundkonzepte von Perl besagt, daß möglichst viel möglich sein soll, aber möglichst wenig zwingend.

Zum Auslesen und Interpretieren von Textdateien bietet Perl durch die Unterstützung regulärer Ausdrücke umfangreiche Möglichkeiten.

Auch die Prüfung und Änderung von Dateiattributen ist in Perl sehr einfach. Von einfach zu verwendenden Dateitestoperatoren (siehe Beispiel in Abbildung 2.5), bis hin zu komplexen Funktionen wie `stat()` sind viele Funktionen implementiert.

Externe Programme können in Perl auf vielfältige Weise aufgerufen werden:

- die Funktion `system()`, die ein Programm startet und auf dessen Beendigung warten
- die "Backtick"-Operatoren, die ein Programm starten und dessen Ausgabe an eine Variable übergeben (z.B. `$file_list=‘ls -l‘;`)
- auf beiden Seiten einer unbenannten Pipe mit `open()`, z.B.:

```
#!/usr/bin/perl -w

$file='sample.pl';

if (-e $file) {
  # file exists
  if (-x $file) {
     print "you are allowed to execute $file\n";
  } else {
     print "you are not allowed to execute $file\n";
  }
}
```

Abbildung 2.5: Beispiel eines *Perl*-Programms zur Prüfung von Dateiattributen

   – `open(FH, "ls -l|");`
   – `open(FH,"|dd of=sample.dat");`)

Zum Aufbau komplexer Datenstrukturen bietet Perl Listen (ähnlich den Listen in Lisp, in Perl auch Arrays genannt), Hashes (entsprechen Arrays in C oder Pascal) und Referenzen (entsprechen Zeigern in C oder Pascal).

Die Code-Dokumentation kann durch das zum Perl-Paket gehörende Programm `perldoc` aufbereitet werden, auch eine Umwandlung in z.B. LaTeX oder HTML ist möglich.

Java

Java ist eine objektorientierte, plattformunabhängige Programmiersprache. Java-Compiler erzeugen einen plattformunabhängigen Bytecode, der von einem Interpreter, der Java Virtual Machine, auf die Maschinensprache der jeweiligen Hardware umgesetzt wird. Interpreter sind für fast alle gängigen Betriebssysteme verfügbar.

Texte können über die Methoden der Klassen `String` und `StringBuffer` untersucht werden. Klassen zur Unterstützung regulärer Ausdrücke sind nicht vorhanden.

Externe Programme können mittels `getRuntime().exec("<prg_call>")` aufgerufen werden, der Ein- und Ausgabestrom des gestarteten Programms kann kontrolliert werden.

Auf C-ähnliche Zeiger wird verzichtet, Arrays, Listen und komplexere Datenstrukturen können aber mit Hilfe von Objekten erzeugt werden. Der Garbage Collector gibt automatisch nicht mehr genutzte Objekte frei, der Entwickler muß sich nicht um die Speicherverwaltung kümmern.

Eine Programm- und Klassendokumentation im HTML-Format kann mittels `javadoc` automatisch erstellt werden. Anders als bspw. `perldoc` fügt `javadoc` auch Informationen aus dem Quellcode selbst (z.B. Aufrufparameter von Methoden) automatisch in die Dokumentation ein.

Mehr Information zu Java und Java-Klassen sind in [Kno97] und [Gra97] zu finden.

C

C wurde von Dennis Ritchie (siehe [BWK88] und [Tan95]) zur Entwicklung des UNIX-Systems entwickelt und kann heute auf praktisch allen Plattformen genutzt werden. Da die Grundfunktionalität von C sehr eingeschränkt ist, werden Funktionsbibliotheken verwendet, die Standardfunktionalitäten zur Verfügung stellen. Leider unterscheiden sich die Bibliotheken von Inhalt und Benennung zwischen einzelnen Betriebssystemen, so daß eine portable Entwicklung zwar möglich, aber ungleich schwieriger als in Perl oder Java zu realisieren ist. Ein weitere Nachteil ist, daß C-Programme auf jeder Ziel-Plattform neu übersetzt werden müssen.

Textdateien können über die im Standard-Modul `string` vorhandenen Funktionen bearbeitet werden (z.B. Teiltextsuche oder Vergleiche), reguläre Ausdrücke werden nicht unterstützt.

Zur Prüfung und Änderung von Dateiattributen gibt es die Funktionen `stat()` und `chmod()` und einen Satz von Makros, mit welchen die Zugriffsberechtigungen geprüft bzw. erzeugt werden können.

Externe Programme können über die Funktionen `exec()` und `system()` oder Abwandlungen von diesen gestartet werden.

Datenstrukturen können über Strukturen und Zeiger aufgebaut werden, auch Arrays fester Länge werden unterstützt.

Eine Dokumentation im Code ist durch Kommentare zwar möglich, Standardprogramme zur automatischen Aufbereitung oder Regeln zur Dokumentation gibt es keine.

Begründung der Entscheidung für Perl

Aus dem obigen Vergleich wird deutlich, daß das beschriebene System wahrscheinlich am einfachsten in Perl implementiert werden kann. In der Funktionalität zur Prüfung von Dateiattributen und zum Aufruf externer Programme reichen weder Java noch C an Perl heran.

Zur Interpretation der Konfigurations- und Ergebnisdateien bietet Perl als einzige der besprochenen Sprache die Unterstützung regulären Ausdrücke.

Einzig in der Art und Weise der Dokumentation hat Java mehr zu bieten, was aber die anderen Nachteile nicht aufwiegen kann.

2.9 Zugriffstransparenz durch erweiterten NFS-Dämon

Wie in der Aufgabenstellung schon angedeutet, soll die Zugriffstransparenz über einen angepaßten NFS-Dämon erreicht werden.

Der Dämon hat dabei die folgenden Aufgaben:

1. Erkennen eines Zugriffs auf eine archivierte Datei und, wenn möglich, Verzögerung dieses Zugriffs bis die Datei wiederhergestellt ist.
2. Anstoßen der Wiederherstellung einer archivierten Datei, wenn darauf zugegriffen wird.

Da die Entwicklung eines NFS-Dämons nicht Teil dieser Arbeit ist und als Referenzplattform Linux benutzt wird, wird ein bestehender Dämon für Linux verändert.

Unter Linux gibt es zur Zeit zwei verschiedenen NFS-Dämonen:

NFS-Server: von Olaf Kirch und anderen entwickelt, ein Dämon der im Benutzermodus läuft, also als ganz normale Anwendung, ohne Einfluß auf den Kernel, die neuste der Version dieses Servers liegt unter [Kir] im Internet

Kernel NFSD: ist Bestandteil des Linux-Kernels, verschiedene Linux-Kernel sind im Internet unter [Tor] zu finden

Da sich der *Kernel-NFSD* zu Beginn dieser Arbeit noch in einem instabilen Zustand befindet, wird der *NFS-Server* verwendet.

Kapitel 3

Programmdesign

Nachdem im letzten Kapitel die Grobstruktur des Programms festgelegt wurde, soll im Folgenden das Design verfeinert und konkretisiert werden.

Dazu werden zunächst die zu implementierenden Regeln (Bedingungen und Aktionen) festgelegt, anschließend wird die Struktur des Dämon-Programms entwickelt. Weiter wird festgelegt, welche Funktionen mit `root`-Berechtigungen ausgestattet sein müssen und daher als `sudo`-Skripte zu implementieren sind. Im Anschluß wird die Schnittstelle zu den Gerätetreibern definiert und die notwendigen Änderungen am verwendeten NFS-Dämon untersucht.

3.1 Aktionen und Bedingungen

Um die Regeln definieren zu können, werden zunächst die Zustände, die eine Datei annehmen kann festgelegt:

zu registrieren: Die Datei wurde im Client-Verzeichnis abgelegt, sie ist vollständig und noch nicht registriert.

registriert: Die Datei wurde vom System erkannt, eine Zuordnung zu Medien hat stattgefunden, die (Teil-) Kopien wurden in die Medienverzeichnisse kopiert.

zu sichern: Die Datei im Medienverzeichnis wurde noch nicht auf einem Medium gesichert.

gesichert: Die Datei ist bereits auf einem Medium gesichert.

wiederherzustellen: Die Datei ist gesichert und in ein Sparsedatei umgewandelt worden. Sie wird benötigt und soll deshalb vom Medium wiederhergestellt werden.

wiederhergestellt: Die Datei ist lesbar im Medienverzeichnis vorhanden. Ein Sonderfall ist der Fall, daß die Datei noch gar nicht gesichert wurde.In diesem Fall ist genaugenommen die Bezeichnung “wiederhergestellt” nicht korrekt, das Ergebnis (die Datei ist lesbar) ist aber das gleiche.

zusammenzusetzen: Die Datei im Client-Verzeichnis ist zu restaurieren, die (Teil-) Kopien wurden schon markiert.

markieren: Die Datei im Client-Verzeichnis ist zu restaurieren, die (Teil-) Kopien wurden noch nicht zur Wiederherstellung markiert.

restauriert: Die Datei im Medien-Verzeichnis wurde vollständig wiederhergestellt und ist für den Benutzer lesbar.

gesparsed: Die Datei ist ein Sparsedatei und enthält momentan keine Daten, außer der Größeninformation.

zu sparsen: Auf eine wiederhergestellte Datei wurde eine bestimmte Zeit lang nicht zugegriffen. Um Plattenplatz zu sparen soll diese Datei in ein Sparsedatei umgewandelt werden.

zu löschen: Eine Datei auf die sehr lange nicht mehr zugegriffen wurde ist vollständig aus dem System zu entfernen. Das bedeutet, daß die Sparsedatei im Client-Verzeichnis, die zugehörige Verwaltungsinformation und die (Teil-) Kopien in den Medienverzeichnissen entfernt werden. Die archivierten Daten auf den Medien bleiben erhalten.

Die Zustände “zu registrieren” und “markieren” werden durch Eingriffe des Benutzers erzeugt. Eine neu abgelegte Datei erhält den Zustand “zu registrieren”, eine Datei, die der Benutzer restaurieren möchte, wird in den Zustand “markieren” versetzt.

Aus diesen Zuständen folgen im Programmverlauf die Zustände “zu sichern”, “wiederherzustellen” und “zu restaurieren” auf den vom Benutzer veränderten Dateien oder auf mit diesen im Zusammenhang stehenden Dateien.

Gänzlich ohne Eingriff des Benutzers können die Dateien die Zustände “zu sparsen” und “zu löschen” erreichen. Diese Zustände werden nach Ablauf bestimmter Fristen für den Datenzugriff erreicht.

Abbildung 3.1 zeigt eine Tabelle mit den Regeln der Zustandsübergänge die vom Programm durchzuführen sind.

Zu beachten ist, daß eine Datei unter Umständen mehrere dieser Zustände gleichzeitig annehmen kann. Eine Datei die sich im Zustand “zu sichern” befindet muß z.B. zwangsläufig auch den Zustand “wiederhergestellt” haben, da eine zu sichernde Datei nicht wiederhergestellt werden kann bevor sie gesichert ist.

3.2 Das Dämon-Programm

Das Dämon-Programm hat hauptsächlich die Aufgabe, die im letztem Abschnitt definierten Regeln umzusetzen und die entsprechende Zustandsübergänge zu realisieren.

Als Hauptprozeß des Programms kommt ihm zusätzlich die Aufgabe der Ressourcenverwaltung zu. Zur Prüfung der Bedingungen muß dem Dämon bekannt sein, welche Geräte im System existieren und auf welche Medien in welcher Form (lesend oder schreibend) zugegriffen werden kann.

Informationen über vorhandene Geräte und deren Fähigkeiten entnimmt der Dämon aus der Konfigurationsdatei. Der momentane Zustand der Geräte wird über den Aufruf der Gerätetreiber ermittelt. Die aktuellen Zustandsdaten werden in einer Datenstruktur im Hauptspeicher gehalten. Da diese Daten wahrscheinlich nach einem Neustart des Programms sowieso nicht mehr aktuell wären, ist eine externe Sicherung in einer Datei nicht notwendig. Die Zustände der Geräte werden einfach erneut geprüft.

Bedingung	Aktion
$\exists$ eine Datei im Zustand "zu sichern", es ist der konfigurierte Sicherungszeitraum, das betreffende Medium ist eingelegt, das Medium ist beschreibbar, $\exists$ ein freies Gerät zum Schreiben des Mediums	Datei auf Medium übertragen, Datei in den Zustand "gesichert" überführen
$\exists$ eine Datei im Zustand "zu restaurieren" $\exists$ eine vollständig wiederhergestellte Kopie dieser Datei	Datei zusammenfügen, in den Zustand "restauriert" überführen
$\exists$ eine Datei im Zustand "markieren"	eine Kopie auswählen, Dateien in Medien-Vz. in Zustand "wiederherzustellen" überführen, Datei im Client-Vz. in den Zustand "zu restaurieren" überführen
$\exists$ eine Datei im Zustand "wiederherzustellen", das betreffende Medium ist eingelegt, $\exists$ ein freies Gerät zum Lesen des Mediums	Datei wiederherstellen und in den Zustand "wiederhergestellt" überführen
$\exists$ eine Datei im Zustand "zu registrieren", $\nexists$ Möglichkeit diese Datei zu registrieren ohne neue Medien zu labeln,	Initialisiere neue Medien
$\exists$ eine Datei im Zustand "zu registrieren", $\exists$ eine Möglichkeit diese Datei zu registrieren ohne neue Medien zu labeln,	Erzeuge eine Zuordnungstabelle (Teil-) Kopie $\rightarrow$ Medium, erzeuge die (Teil-) Kopien in den entspr. Medien-Vz., überführe diese (Teil-) Kopien in den Zustand "zu sichern", überführe die Originaldatei in den Zustand "registriert"
$\exists$ eine Datei im Zustand "zu sparsen"	Datei in Sparsedatei umwandeln
$\exists$ eine Datei im Zustand "zu löschen"	Datei in löschen

Abbildung 3.1: Vom Programm zu realisierende Zustandsübergänge einer Datei

FASY-Daemon — Der Dämon-Hauptprozeß

```
lese Konfigurationsdatei
erzeuge Datenstruktur zur Geräteverwaltung
while(TRUE)
    aktualisiere Gerätestatus
    durchsuche Verzeichnisse und kategorisiere Dateien
    prüfe Regeln, führe Aktionen aus
    kurze Pause, dann weitermachen
```

Abbildung 3.2: Nassi-Schneiderman-Diagramm des Dämon-Hauptprozesses

Zum Prüfen der Bedingungen ist es notwendig sowohl die Client-, als auch die Medien-Verzeichnisse nach Dateien in bestimmten Zuständen zu durchsuchen. Um eine wiederholte, zeitraubende und Hardware belastende Suche auf den gleichen Dateien zu vermeiden, werden die betreffenden Verzeichnisse vor Beginn der Bedingungsprüfungen durchsucht und die enthaltenen Dateien entsprechend kategorisiert.

Die Aufgaben des Dämons lassen sich wie folgt zusammenfassen:

- Einlesen der Konfigurationsdatei und erstellen der Datenstruktur zur Verwaltung der Sicherungsgeräte.
- Sicherstellen der Aktualität der Gerätezustände in der Datenstruktur.
- Durchsuchen der Client- und Medien-Verzeichnisse und kategorisieren der enthaltenen Dateien.
- Durchführen der beschriebenen Regeln zum Zustandsübergang der Dateien.

Der erste Punkt muß lediglich einmal beim Programmstart ausgeführt werden. Die anderen Punkte sollten periodisch (z.B. in einer Schleife) bearbeitet werden, denn sowohl der Zustand der Geräte als auch der Zustand der Dateien kann sich jederzeit ändern.

Zur Senkung der Systembelastung "legt" sich das Programm nach jedem Schleifendurchlauf für eine bestimmte Zeit "schlafen".

Das Nassi-Schneiderman-Diagramm in Abbildung 3.2 zeigt die grundsätzliche Funktionsweise des Dämon-Programms.

3.2.1 Datenverwaltung

Bisher ist weitgehend offen geblieben wie die anfallenden Daten (z.B. Dateizustand, Zuordnungstabelle *Datei* $\rightarrow$ *Medien*) tatsächlich gesichert werden. Diese Frage wird in den nächsten Abschnitten geklärt.

Zustandsbeschreibung einer Datei

Wie schon mehrfach erwähnt, wird der Zustand einer Datei über deren Zugriffsberechtigungs-Flags angezeigt. Diese Flags werden von Benutzern normalerweise dazu verwendet den Zugriff auf die Datei für andere Benutzer freizugeben oder einzuschränken. Könnten die Benutzer im Archivierungssystem die Flags der Dateien selbst setzten, wären diese als Zustandsinformation unbrauchbar.

Die Benutzer können nur dadurch wirkungsvoll daran gehindert werden die Flags einer Datei zu verändern, wenn sie nicht Eigentümer der Datei sind. Als neuer Eigentümer dieser Datei bietet sich der Benutzer des Archivierungssystems an.

Um den Benutzern weiterhin den lesenden Zugriff auf ihre Daten zu ermöglichen, müssen alle Dateien für sämtliche Benutzer lesbar sein. Dies führt zu der Frage, wie man archivierte Dateien trotzdem vor dem Zugriff anderer Benutzer schützen kann. Die Antwort besteht darin, die zu schützenden Dateien in einem Unterverzeichnis abzulegen, auf welches nur der Eigentümer des Verzeichnisses zugreifen darf.

Der Zustand einer Datei hängt also nicht mehr nur von deren Flags, sondern auch von deren Eigentümer ab. Der Archivierungsbenutzer wird zum Eigentümer einer Datei, sobald diese erfolgreich registriert worden ist.

Zusammenfassend läßt sich über den Zusammenhang Datei-Flags, Datei-Besitzer und Dateizustand folgendes sagen:

- eine Datei, die nicht dem Archivierungsbenutzer gehört, befindet sich im Zustand “zu registrieren”
- nach erfolgreicher Registrierung geht die Datei in den Besitz des Archivierungsbenutzers über
- der Zustand der Dateien, die dem Archivierungsbenutzer gehören, kann an deren Flags abgelesen werden

Besonderheit des Zustands “zu sichern”

Der Zustand “zu sichern” kann nicht in den Dateiflags abgelegt werden, da in den Medienverzeichnissen Hardlinks der zu sichernden Datei abgelegt sein können. Soll nun eine Datei auf mehreren Medien gesichert werden, werden mehrere Hardlinks erzeugt. Würde eine dieser “Kopien” auf ein Medium kopiert und als “gesichert” markiert, wären gleichzeitig auch alle anderen Kopien als “gesichert” markiert, da sie alle auf die gleiche Datei verweisen.

Darum muß in den Medien-Verzeichnissen vermerkt werden, welche Dateien bereits gesichert sind und welche nicht. Dazu wird in jedem Medien-Verzeichnis eine Datei angelegt, welche die Namen aller bereits gesicherten Dateien enthält. Nach einer erfolgreichen Sicherung wird diese Datei um die Namen der neu gesicherten Dateien erweitert.

Diese Aktion kann mittels der beschriebenen Methode zur “atomaren” Erzeugung von Dateien einen Transaktionscharakter bekommen.

Auswahl einer bestimmten Kopie zur Restauration

Zur Markierung einer Datei als “zu restaurierend” genügt ein Flag. Ist eine Datei allerdings mehrfach gesichert und der Benutzer möchte gezielt auswählen welche Sicherung wiederherzustellen ist, ist eine andere Technik notwendig.

Da die Zahl der möglichen Kopien einer Datei nicht beschränkt werden soll, ist eine Technik zu finden, die eine Auswahl aus einer großen Menge von Kopien ermöglicht.

Eine Möglichkeit, die gut in das bisherige Schema der Dateiflags paßt, ist die Datei durch einen symbolischen Link zu ersetzen. Wird der Verweis auf die wiederherzustellende Kopie als Ziel des Links angegeben, steht eine ausreichend große Menge von Auswahlmöglichkeiten zur Verfügung.

Das Programm wählt die herzustellende Kopie aus, indem das Ziel des symbolischen Links und die Pfade zu den Kopien verglichen werden. Die erste Kopie, deren Pfad das Ziel des Links enthält, wird gewählt. Der Pfad zu einer Kopie enthält u.a. auch den Namen der Serie. Erzeugt der Benutzer einen Link, dessen Ziel den Namen einer Serie enthält, so wird die Kopie auf dieser Serie wiederhergestellt.

Um dem NFS-Dämon die Verwirklichung des transparenten Zugriffs zu ermöglichen, wird ebenfalls eine Markierung zu restaurierender Dateien über Flags unterstützt. Wird eine Restauration über Flags angestoßen, wird immer die erste Kopie wiederhergestellt.

Serien- und Medienverzeichnisse

Eine der neuen Ideen im Archivierungssystem ist die Verwendung von Serien. Im Abschnitt 2.1.2 wurde bereits definiert, daß eine Serie eine Menge von zusammengehörenden Medien ist.

Um den Medienverzeichnissen auf der Holding-Disk eine gewisse Struktur zu geben, werden Serienverzeichnisse eingeführt. Sämtliche Medienverzeichnisse einer Serie werden in deren Serienverzeichnis abgelegt.

Zur einfachen Handhabung werden die Serienverzeichnisse nach der Serienkennung und die einzelnen Medienverzeichnisse nach der Kennung der zugehörigen Medien benannt.

Verwaltung der Zuordnungsdaten

Zu jeder registrierten Datei gibt es eine Zuordnung zu einer oder mehreren Dateien auf einem oder mehreren Medien, auf denen die Datei archiviert wird. Um die Datei im Restaurationsfall wieder finden zu können, müssen diese Daten irgendwo persistent gesichert werden.

Eine persistente Sicherung von Daten kann in Dateien erfolgen, dazu gibt es zwei unterschiedliche Ansätze:

1. Sämtliche Daten werden zentral in einer einzigen Datei gehalten.
2. Die Daten werden verteilt bei den betreffenden Dateien abgelegt.

Die erste Variante hat den Vorteil, daß nur eine einzige Datei existiert und somit der Speicherplatz besser ausgenutzt wird. (Kleine Dateien belegten mindestens einen Inode, auch wenn sie nur einen Bruchteil des Platzes benötigen.) Nachteilig ist, daß entweder alle oder gar keine Zuordnungen für die Benutzer einsehbar sind, je nachdem ob er die Datei lesen darf oder nicht. Da das Wissen über die Zuordnung von Dateien anderer Benutzer ein Sicherheitsrisiko bedeutet, soll die Einsicht in diese Datei versperrt bleiben. Daraus ergibt sich für die Benutzer der Nachteil, daß sie nicht nachvollziehen können, wo ihre Daten abgelegt werden.

```
Serie1/Serie1_0001/file0002 Serie1/Serie1_0002/file0001
SerieXX/SerieXX_0011/file0005
```

Abbildung 3.3: Beispiel einer Referenzdatei

In der zweiten Variante gibt es für jede registrierte Datei eine Referenzdatei im gleichen Verzeichnis. In dieser Referenzdatei ist die Zuordnungstabelle abgelegt. Der Benutzer kann auf diese Art verfolgen, wo seine Daten abliegen und, wenn mehrere Kopien erstellt wurden, bei der Restauration zwischen diesen wählen.

Der Nachteil dieser Variante ist, daß der Namensraum der zu archivierenden Dateien eingeschränkt wird, da die Referenzdateien auf keinen Fall archiviert werden dürfen (sonst fehlt der Verweis auf die Daten). Da aber auch die benutzerdefinierten Einstellungen (Seriendefinitionen etc.) über Dateien realisiert werden, ist ohnehin eine Einschränkung nötig. Unter UNIX werden die Namen von Konfigurationsdateien der Benutzer üblicherweise mit einem führenden Punkt eingeleitet und sind somit in den meisten Anwendungsfällen verdeckt.

Eine Beschränkung des Namensraums dahingehend, daß Dateien mit führendem Punkt nicht archiviert werden ist also nichts ungewöhnliches. Der Benutzer bekommt dafür die Möglichkeit, die Medien zu bestimmen, auf denen eine Datei abgelegt ist.

Die Zuordnungsdaten der archivierten Dateien werden in einer Referenzdatei abgelegt, deren Namen mit einem Punkt beginnt und durch eine festgelegte Regel aus dem Namen der archivierten Datei erzeugt werden kann.

Der Aufbau der Referenzdatei wird möglichst einfach gewählt:

- jede Zeile repräsentiert eine Kopie
- jede Zeile enthält einen Verweis auf die zugehörige Kopie im Medien-Verzeichnis
- besteht eine Kopie aus mehreren Teilkopien, so werden Verweise auf diese Teilkopien in der gleichen Zeile durch Leerzeichen getrennt

Die Referenzdatei zu einer Datei `sample.tgz`, die in den Serien `Serie1` und `SerieXX` gesichert wird, könnte z.B. wie in Abbildung 3.3 aussehen.

Man sieht, daß die erste Kopie dieser Datei auf zwei Medien (`Serie1_0001` und `Serie1_0002`) aufgeteilt wurde, die zweite Kopie wurde vollständig auf dem Medium `SerieXX_0011` der Serie `SerieXX` gesichert.

3.2.2 Benutzerdefinierte Konfiguration

Neben den systemweiten Einstellungen, die vom Systemadministrator vorzunehmen sind, werden auch den Benutzern bestimmte Rechte eingeräumt. Im Gegensatz zu den globalen Einstellungen können die Benutzereinstellungen nicht in einer globalen Datei vorgenommen werden, sonst könnten die Benutzer nicht nur ihre eigenen, sondern auch die Einstellungen der anderen Benutzer, ändern.

Um den Benutzer zusätzlich die Möglichkeit zu geben, für jedes Unterverzeichnis andere Einstellungen zu verwenden, werden die Konfigurationsdateien in den Verzeichnissen selbst erstellt.

```
*.tgz *.tar.gz : TGZ_SERIE, ALLGEMEINE_SERIE + DEFAULT_SERIE
*.jpg : MEINE_BILDER1, MEINE_BILDER_DEFAULT | BILDER_BACKUP
* : DEFAULT_SERIE
```

Abbildung 3.4: Beispiel einer Assoziationsdatei

Die in Kapitel 2.6.2 beschriebenen Einstellungsmöglichkeiten werden dabei wie folgt realisiert:

Die Assoziationsdatei

In dieser Datei wird festgelegt, welche Dateitypen auf welchen Serien zu archivieren sind. Der Benutzer soll die Möglichkeiten haben, seine Datei zur Sicherheit auf mehrere Medien zu sichern und Alternativen anzugeben. (Zum Beispiel für den Fall, daß für eine Serie zur Zeit keine Medien zur Verfügung stehen.)

Es wird folgender Dateiaufbau gewählt:

Jede Assoziation wird in einer Zeile beschrieben: Nach der Angabe einer oder mehrerer Masken der zu archivierenden Dateien folgt ein Doppelpunkt und ein Term der Serien auf denen gesichert werden soll. Eine Dateimaske kann, wie unter UNIX üblich, mit `?` und `*` angegeben werden (z.B. `*.tgz`).

Ein Term von Serien kann wie folgt aufgebaut sein:

- ein Serienname
- mehrere Serienterme durch den Operator “`,`” verbunden
- mehrere Serienterme durch den Operator “`|`” verbunden
- mehrere Serienterme durch den Operator “`+`” verbunden

Der Operator “`,`” bezeichnet eine Alternative, z.B. bedeutet `Serie1, Serie2` folgendes: “Sichere auf `Serie1`, wenn das möglich ist, sonst auf `Serie2`”. Eine Sicherung auf einer bestimmten Serie gilt dann als möglich, wenn genügend Platz auf den bereits initialisierten Medien vorhanden ist oder wenn dieser Platz durch labeln verfügbarer leerer Medien bereitgestellt werden kann.

“`|`” ist eine abgeschwächte Form von “`,`”. Ein Term `Serie1 | Serie2` würde bedeuten “Sichere auf `Serie1`, wenn dies möglich ist, ohne neue Medien zu labeln, sichere sonst auf `Serie2`”.

Zusätzlich gibt es noch den UND-Operator “`+`”, der besagt, daß eine Sicherung auf beiden angegebenen Serien erfolgen soll.

Die Bindungskraft der Operatoren nimmt in der Liste von oben nach unten zu. Ein Beispiel für eine Assoziationsdatei ist in Abbildung 3.4 zu sehen.

Die Seriendatei

Die Seriendatei gibt dem Benutzer die Möglichkeit eigene Serien zu definieren, die in den Assoziationsdateien zur Bildung von Assoziationen verwendet werden können.

```
serie BEISPIEL = {
        media_types=dds3
        location=UNI_STUTTGART_IFI
        write_only_when_full=FALSE
}
```

Abbildung 3.5: Beispiel einer Seriendefinition

Abbildung 3.5 zeigt ein Beispiel einer Seriendefinition. Zur Definition einer Serie sind folgende Parameter notwendig:

- Art der zur Sicherung zu verwendenden Medien (Parameter `media_types`)
- Lokation der Serie (Parameter `location`), es werden für diese Serie nur Medien gelabelt, die in Geräten mit einer der angegebenen Lokationen liegen.
- Sicherungsstrategie (Parameter `write_only_when_full`): Soll das Medium erst beschrieben werden, wenn sämtliche darauf zu sichernden Dateien im Medien-Verzeichnis abgelegt wurden oder soll eine Sicherung auch schon früher möglich sein?

3.2.3 Fehlerbehandlung

Dämon-Programme haben, wie bereits erwähnt, keine Möglichkeit auftretende Fehler interaktiv mitzuteilen. Trotzdem ist es gerade bei Dämon-Programmen wichtig, daß Fehler oder Statusinformationen an den Systemadministrator gemeldet oder zumindest in einer Logdatei protokolliert werden. Der Systemadministrator soll mithilfe dieser Dateien in die Lage versetzt werden, aufgetretene Probleme zu lokalisieren und eventuell Maßnahmen zur Fehlerbehebung einzuleiten.

Aber auch, wenn keine Fehler auftreten, ist es für Dämonen manchmal nötig dem Administrator irgendwelche Mitteilungen zu machen. Das zu entwickelnde Archivierungssystem benötigt den Administrator z.B. zum Wechseln von Medien für Restaurations- und Sicherungsaktionen. Die Aufgabenstellung verlangt, daß der Administrator per E-Mail zu benachrichtigen ist, wenn ein manuelles Eingreifen erforderlich wird. Dies hat für den Administrator den Vorteil, daß sich das System bei Problemen von sich aus meldet. Es ist kein besonderes Programm zur Erkennung von Fehlern und Problemen notwendig.

Ein erster Ansatz führt zu der Idee, immer dann, wenn ein Fehler auftritt oder aus irgendeinem anderen Grund ein Administratoreingriff gefordert ist, eine E-Mail abzusenden. Ein Systemadministrator würde aber wenig erfreut über solch eine Implementierung sein. Er bekäme zwar nach jedem Fehler eine Benachrichtigung, könnte sich aber vermutlich vor Fehlermeldungen kaum retten.

Man stelle sich z.B. eine Situation vor, in der eine große Zahl von Dateien, z.B. 50, gleichzeitig wiederhergestellt werden soll. Einige dieser Dateien (z.B. 10) können vermutlich von eingelegten Medien wiederhergestellt werden, es bleibt aber wahrscheinlich eine größere Anzahl von Dateien (im Beispiel 40) übrig, die von momentan nicht eingelegten Medien hergestellt werden müssen.

Das System prüft nacheinander sämtliche wiederherzustellende Dateien und stellt dabei fest, daß jeweils ein bestimmtes Medium einzulegen ist. Dies wird dann dem

Administrator per E-Mail mitgeteilt und die nächste Datei wird geprüft. Der Administrator erhält auf diese Weise 40 E-Mails, mit der Aufforderung ein bestimmtes Medium einzulegen. Vermutlich würde sich kein Administrator die Mühe machen alle diese Nachrichten durchzulesen.

Noch verschärft wird die Situation, wenn der Administrator momentan nicht sofort reagieren kann und das Archivierungssystem in einem nächsten Durchlauf (z.B. 5 Minuten später) wieder 40 E-Mails verschickt und nach weiteren 5 Minuten wieder usw.

In einer effektiveren Methode werden alle Fehlermeldungen über einen bestimmten Zeitraum gesammelt und am Ende einer Periode in einer einzigen E-Mail zusammen verschickt. Die Länge einer solchen Periode kann der Administrator je nach Bedarf einstellen. Werden in den so entstehende Benachrichtigungen noch die mehrfach auftauchenden Meldungen entfernt, bekommt der Administrator einen übersichtlichen Statusbericht.

Zur einfacheren Programmierung wird die gesamte Fehlerbehandlung (Schreiben in die Protokolldatei und Administratorbenachrichtung) vereinheitlicht. Fehler werden über einen Funktionsaufruf bekanntgemacht, in eine Logdatei geschrieben und, je nach Konfiguration, an den Administrator weitergeleitet.

Der Administrator möchte natürlich nicht sämtliche Aktionen gemeldet bekommen, andererseits sollte die Protokolldatei relativ aussagekräftig sein. Dies wird durch folgende Konstruktion erreicht:

- Fehler werden in verschiedene Klassen eingeteilt:

  fatal: schwerwiegende Fehler, die einen Programmabbruch oder den Abbruch eines Teilprogramms zur Folge haben

  nonfatal: weniger schwere Fehler, das Programm kann weiterarbeiten

  info: Informationen an den Administrator

  trace: Ablaufinformationen, können zur Fehlersuche eingesetzt werden

- In der Konfigurationsdatei kann festegelegt werden, welche Art von Fehlern an den Administrator weitergeleitet und/oder in die Protokolldatei geschrieben werden.
- In der Protokolldatei werden sämtliche Fehlermeldungen sofort und mit der Uhrzeit des Auftretens protokolliert.
- Die maximale Größe der Protokolldatei ist einstellbar, ist die Datei zu groß, wird sie gelöscht und neu geschrieben.
- Fehler, die an den Administrator gemeldet werden, werden zunächst gesammelt und periodisch versendet.
- Die E-Mails an den Administrator beinhalten zusätzlich eine Aufstellung von Fehlern, die seit der letzten Statusmeldung nicht mehr, wieder oder neu aufgetreten sind.

3.3 SUDO-Skripte

Das Dämon-Programm läuft unter dem Benutzer des Archivierungssystem und unterliegt daher den gleichen Zugriffsbeschränkungen wie jeder andere Benutzer. Für einige Aktionen sind jedoch weiterreichende Berechtigungen notwendig. Mit welchen

Methoden eine solcher Erweiterung der Berechtigungen möglich ist, wurde schon im Kapitel 2.5 erläutert. In diesem Abschnitt soll nun geklärt werden, welche Aktionen mit `root`-Berechtigung ausgeführt werden müssen.

In Frage kommen nur Aktionen, die im Client-Verzeichnis stattfinden. Sämtliche anderen Verzeichnisse sollten dem Archivierungsbenutzer selbst gehören, weshalb er in diesen Verzeichnis zu fast jeder Aktion berechtigt ist[1].

In den Client-Verzeichnissen sind aber alle Aktionen als `root` auszuführen, darunter fallen folgende:

- durchsuchen der Verzeichnisse nach Dateien und prüfen der Datei-Flags
- auslesen der Assoziations- und Serienkonfigurationsdateien
- schreiben und auslesen der Referenzdateien
- kopieren der Dateien von oder in Client-Verzeichnisse
- löschen und "sparsen" von Dateien

Um die Anzahl der nötigen Skripte gering zu halten wird ein "Wrapper"-Skript aufgerufen, welches den Namen einer Funktion und deren Parameter übergeben bekommt und das Ergebnis zurückgibt.

Dieses Skript reicht aus, um Funktionen mit einfachen Rückgabewerten, wie z.B. Löschen oder Kopieren von Dateien, auszuführen. Zum Auslesen und Schreiben von Dateien wird ein anderes Verfahren verwendet:

Es wird ein SUDO-Skript gestartet, welches über eine Pipe mit dem Dämon-Programm verbunden ist. Mit entsprechender Kapselung in Modulen kann so erreicht werden, daß es für das Dämon-Programm aussieht, als könne die Datei ganz normal gelesen bzw. geschrieben werden.

Die Zahl der zu implementierenden SUDO-Skripte läßt sich also auf drei reduzieren:

- ein "Wrapper"-Skript zur Ausführung von Funktionen mit einfachen Rückgabewerten
- ein Skript zum Lesen von Dateien
- ein Skript zum Schreiben von Dateien

Damit wird der Archivierungsbenutzer natürlich zu einem sehr großen Sicherheitsrisiko, denn er kann über das "Wrapper"-Skript sämtliche Perl-Operationen ausführen. Deshalb sollte das Passwort des Archivierungsbenutzers nur dem Systemadministrator, der sowieso `root`-Berechtigung besitzt, bekannt sein. Alternativ könnte der Benutzer des Archivierungssystem auch für die Anmeldung allgemein gesperrt werden, indem das Paßwort in der Datei `/etc/passwd` als ungültig markiert wird. In diesem Fall kann nur der `root`-Benutzer die Identität des Archivierungsbenutzer erlangen (z.B. über den Befehl `su`).

3.4 Gerätetreiber

Die Schnittstelle zu den Gerätetreibern sollte möglichst flexibel gestaltet sein, um eine große Zahl von Geräten unterstützen zu können. Gleichzeitig sollte sie relativ einfach gehalten werden, damit Gerätetreiber schnell und ohne großen Aufwand erstellt werden können.

[1]Einige Aktionen, z.B. `chown`, können in allen Verzeichnissen nur von `root` ausgeführt werden.

Zur Definition der Schnittstelle ist zunächst zu überlegen, bei welchen Aktionen ein Zugriff auf ein Gerät notwendig ist. Es sind folgende:

- Ermittlung des Gerätestatus und der eingelegten Medien
- Initialisieren (labeln) von leere Medien
- Übertragen von Dateien auf das Medium
- Wiederherstellen von Dateien vom Medium

Jede der oben genannten Aktionen wird in einem eigenen Programm oder Skript implementiert.

Zu jedem Gerät gibt es ein Verzeichnis, in dem die Gerätetreiber Statusinformationen ablegen können. Jedes der oben genannten Treiberprogramme bekommt u.a. dieses Verzeichnis als Parameter übergeben und kann dort Daten ablegen oder auslesen.

Der komplexeste Teil des Gerätetreibers ist die Gerätestatusprüfung. Dieses Treiberprogramm soll überprüfen, in welchem Status sich die Teilgeräte[2] befinden und prüfen, welche Medien eingelegt sind. Diese Daten werden in Dateien geschrieben, die vom Dämon-Prozeß ausgelesen werden.

Die Gerätestatusprüfung hat auch die Aufgabe der Synchronisation für das zu prüfende Gerät. Sie muß prüfen, welche der Teilgeräte frei und benutzbar sind und diese dem Dämon-Prozeß melden.

Da alle Prozesse, die ein Gerät benutzen, dessen Gerätedatei als Aufrufparameter übergeben bekommen, stellt sich die Synchronisation relativ einfach dar: Es muß lediglich geprüft werden, ob ein anderer Prozeß mit dem Gerät im Kommandozeilenaufruf läuft.

Die Aufgaben der anderen Treiberkomponenten sind oben schon eindeutig geklärt, auf sie wird deshalb hier nicht näher eingegangen.

Eine genaue Beschreibung zur Implementierung von Treibern kann dem Programmhandbuch entnommen werden.

3.5 Erweiterungen/Änderungen am NFS-Dämon

3.5.1 Kurze Einführung in die Arbeitsweise von RPC und NFS

NFS ist ein Netzwerk-Dateisystem mit dem Verzeichnisse entfernter Rechner in den Verzeichnisbaum des lokalen Rechners eingebunden werden können. Die notwendigen Aktionen werden als RPC[3]-Funktionen realisiert.

Kurz gesagt verschickt ein RPC-Funktionsaufruf die übergebenen Parameter an den Server, wo die entsprechende Funktion mit diesen Parametern aufgerufen wird. Das Ergebnis des Funktionsaufrufs wird wieder an den Client-Rechner zurückübermittelt, so daß die Anwendung (außer zeitlichen Verzögerungen) keinen Unterschied zwischen einem RPC-Funktionsaufruf und dem Aufruf einer lokalen Funktion bemerkt.

[2] Ein Sicherungsgerät kann aus mehreren Geräten bestehen, es gibt z.B. CD-Roboter mit mehreren Leseeinheiten

[3] RPC ist die Abkürzung für *remote procedure call*

NFS-Daemon — NFS-Dämon Version 3 - READ-CALL

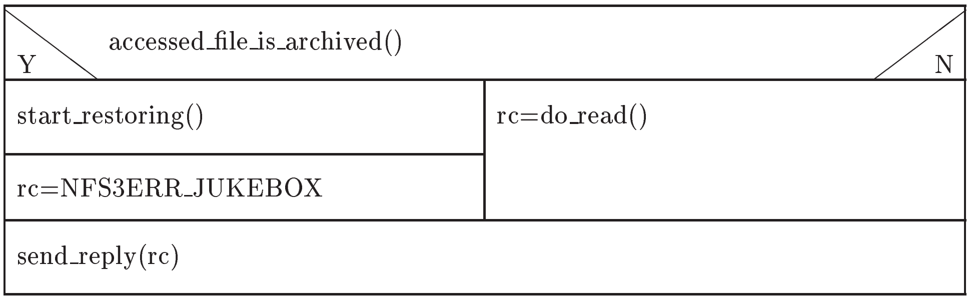


Abbildung 3.6: Lösungsansatz NFS-Dämon Version 3, erweiterte Logik beim Lesezugriff

NFS bildet die dateisystemtypischen Funktionen wie `read()` und `write()` über RPC-Funktionen ab und ermöglicht so den Zugriff auf ferne Dateisysteme.

Eine Einführung in die Funktionsweise von RPC gibt es in [Tan95], Kapitel 10.3, mehr über NFS steht u.a. in den RFCs 1094 (Beschreibung des NFS-Protokolls, Version 2,[Gro89]) und 1813 (Beschreibung der Version 3 des NFS-Protokolls, [Cal95]).

3.5.2 Anpassungs-Ideen

Hinsichtlich des Archivierungssystems ist eine Änderung des NFS-Dämons dahingehend erforderlich, daß beim Öffnen einer Datei deren Rechte überprüft werden und - wenn die Datei sich nicht im Zustand "wiederhergestellt" befinden sollte - deren Restaurationsprozeß angestoßen wird. Der Client sollte bis zur erfolgten Restauration der Datei blockiert bleiben.

Da das NFS-Protokoll keinen expliziten Aufruf zum Öffnen einer Datei kennt, muß hier beim Lesen der Datei angesetzt werden.

Wie eine Blockierung des Clients erzeugt werden kann, hängt von der verwendeten NFS-Version ab.

Lösungsansätze mit NFS Version 3

Seit **NFS Version 3** gibt es den Rückgabewert `NFS3ERR_JUKEBOX`, der dem Client mitteilt, daß die angeforderte Datei momentan nicht verfügbar ist und daß er es später noch einmal versuchen soll (vgl. [Cal95]).

Die Lösung in diesem Fall ist relativ einfach: Wird festegestellt, daß auf eine archivierte Datei zugegriffen werden soll, wird die Restauration der Datei angestoßen und der Wert `NFS3ERR_JUKEBOX` zurückgegeben.

Die zugreifende Client-Anwendung wartet nach Empfang dieses Rückgabewerts einige Zeit und versucht den Zugriff dann erneut.

Abbildung 3.6 verdeutlicht den Lösungsansatz in einem Nassi-Schneiderman-Diagramm.

NFS-Daemon — NFS-Dämon Version 2 - READ-CALL blocking

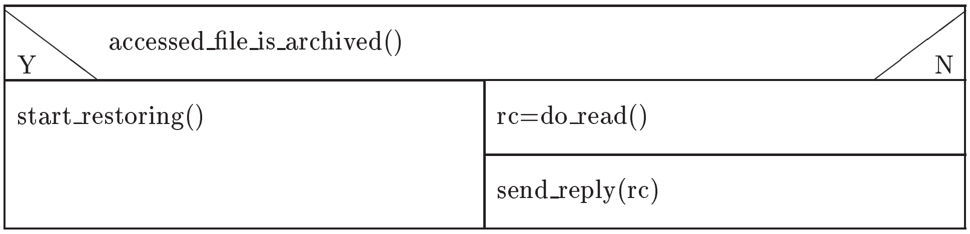


Abbildung 3.7: Blockierender Lösungsansatz NFS-Dämon Version 2

Lösungsansätze mit NFS Version 2

In der häufiger verwendeten früheren **NFS Version 2** existiert ein solcher Rückgabewert nicht. Die Blockierung des Clients muß also auf andere Weise erreicht werden ohne daß Änderungen am Client notwendig sind. Eine Idee ist die Ausnutzung des folgenden Client-Verhaltens:

Bekommen NFS-Clients keine Antwort auf eine Anfrage, so senden sie ihre Anfrage so oft an den Server bis dieser antwortet oder bis ein sogenannter *major timeout* auftritt. Mit entsprechender Konfiguration kann man die *major timeouts* völlig ausschalten, so daß der Client es unendlich oft probiert, bis der Server antwortet.

Ein NFS-2-Server muß also beim Lesezugriff die Rechte einer Datei prüfen und nur in dem Fall eine Antwort zurückschicken, wenn die Datei verfügbar ist. Ist das nicht der Fall, wird eine Restauration angestoßen, dem Client aber keine Antwort gesendet. Daraufhin schickt der Client seine Anfrage immer wieder. Früher oder später wird die Datei restauriert sein, eine Anfrage des Clients kann jetzt beantwortet werden.

Abbildung 3.7 zeigt ein Nassi-Schneiderman-Diagramm dieses Lösungsvorschlags.

Ein Nachteil dieser Methode ist, daß der NFS-Client keine anderen Anfrage sendet, bis diese eine Anfrage beantwortet ist. Dies führt dazu, daß vom blockierten Client-Rechner aus alle Zugriffe auf das gemountete Dateisystem blockiert sind bis die Datei wiederhergestellt ist.

Eine andere Lösung für **NFS Version 2** zeigt Abbildung 3.8. Im Falle eines Zugriffs auf eine lokal momentan nicht vorhandene Datei wird ein Fehlercode zurückgegeben. Der Client meldet daraufhin einen *Ein-/Ausgabefehler* für diese Datei. Diese Methode hat für den Benutzer den Nachteil, daß er nicht Unterscheiden kann, ob ein tatsächlicher Fehler aufgetreten ist oder ob die Restaurations der Datei gestartet wurde.

Beide Lösungen für **NFS Version 2** sind nicht perfekt, der Benutzer soll selbst wählen, welche er verwenden möchte. Darum wird der Dämon so gestaltet, daß er per Aufrufparameter die eine oder die andere Lösung anbietet.

3.5.3 Das Inode-Problem

Wenn eine Anwendung eine Datei in einem per NFS gemounteten Verzeichnis öffnet, bekommt sie wie beim Öffnen einer lokalen Datei ein Dateihandle. Dieses Dateihandle enthält unter anderem den Inode der geöffneten Datei. Ändert sich der

NFS-Daemon — NFS-Dämon Version 2 - READ-CALL nonblocking

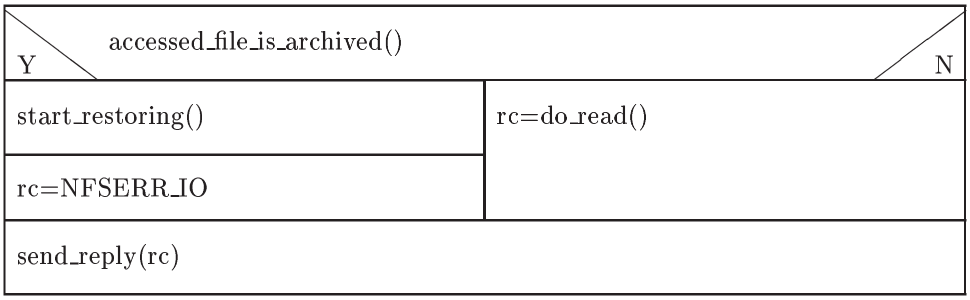


Abbildung 3.8: Nicht blockierender Lösungsansatz NFS-Dämon Version 2

Inode einer Datei während ein Programm versucht daraus zu lesen, wird das NFS-Handle ungültig. Wird eine Datei restauriert, ist darauf zu achten, daß der Inode der Datei gleich bleibt, da das zugreifende Programm auf dem Client-Rechner sonst einen Fehler (`stale nfs-handle` o.ä.) meldet.

Das bedeutet, daß Dateien bei der Restauration nicht über die beschriebenen Logik zur "atomaren" Erzeugung von Dateien geschrieben werden können, da eine neu erzeugte Datei (die zunächst temporäre Datei) einen anderen Inode haben muß als die Originaldatei. Die vorhandene Datei (die Sparsedatei) muß also erhalten bleiben und mit Daten "aufgefüllt" werden. Wird eine bestehende Datei zum Schreiben geöffnet, ändert sich deren Inode nicht.

Dieses Vorgehen stört die transaktionsorientierte Logik aber nicht, da das Programm den Zustand einer Datei an deren Flags erkennt. Eine Datei gilt erst als wiederhergestellt, wenn ihre Flags dies anzeigen. Werden die Flags erst nach erfolgreicher Wiederherstellung gesetzt, ist die atomare Logik weiterhin vorhanden. Dateien die nicht vollständig wiederhergestellt sind, werden als "nicht wiederhergestellt" gesehen.

Kapitel 4

Implementierung

In den vorangegangenen Kapitel wurden Design-Fragen und grundsätzliche Funktionsweisen geklärt. Das folgende Kapitel beschäftigt sich nun mit der Implementierung der entstandenen Lösung in Perl.

Dazu wird zunächst der Aufbau des Dämon-Programms und seiner Module beschrieben. Anschließend werden die Einbindung der SUDO-Skripte in das Programm und die Änderungen am NFS-Dämon erklärt.

4.1 Das Dämon-Programm

Die Struktur des Dämon-Programms ist in Kapitel 3.2 schon ziemlich genau festgelegt worden, die reale Implementierung erweitert dieses Design an wenigen Punkten. An dieser Stelle wird kurz begründet, welchen Sinn die eingebauten Erweiterungen machen.

Ein Nassi-Schneiderman-Diagramm der realen Implementierung in der Programmdatei `fasy.pl` ist in Abbildung 4.1 zu sehen.

Nach dem Einlesen der Konfigurationsdatei werden temporäre Dateien von bisherigen Programmläufen entfernt. Das Löschen der alten temporären Dateien besteht aus zwei Schritten:

1. Dem Löschen der alten Verzeichnisse zur Interprozeßkommunikation. Dieser Schritt ist notwendig, um zu verhindern, daß eventuell noch abliegende alte Dateien interpretiert werden.
2. Dem Löschen der temporär erzeugten Dateien zur Simulierung einer atomaren Dateierzeugung. Dieser Schritt ist zwar nicht zwingend erforderlich, diese Dateien belegen aber unnötig Speicherplatz, der eventuell später gebraucht werden kann.

Anschließend wird aus den Daten der Konfigurationsdatei eine Liste der im System vorhandenen Sicherungsgeräte aufgebaut und der Prozeß in den Hintergrund verlagert. Eine Abkoppelung des Prozesses von der interaktiven Shell wird erst nach Interpretation der Konfigurationsdatei vollzogen, damit im Falle von Konfigurationsfehlern der Benutzer sofort informiert werden kann.

Die automatische Verlagerung des Prozesses in den Hintergrund vereinfacht die Benutzung des Programms und verhindert ungewollte Programmabbrüche z.B.durch

FASY-Daemon — Der Dämon-Hauptprozeß

- lese Konfigurationsdatei
- lösche temporäre Dateien vom letzten Programmlauf
- erzeuge temporäre Verzeichnisse zur Interprozeßkommunikation
- erzeuge Datenstruktur zur Geräteverwaltung
- Prozeß in den Hintergrund verlagern (als Dämon weiterlaufen)
- Liste der Bedingungen und Aktionen aufbauen
- while(TRUE)
  - Dateilisten löschen
  - aktualisiere Gerätestatus
  - durchsuche Verzeichnisse und kategorisiere Dateien in Dateilisten
  - prüfe Regeln, führe Aktionen aus
  - bei Empfang des HUP-Signals sofort weitermachen, sonst kurze Pause

Abbildung 4.1: Nassi-Schneiderman-Diagramm des realen Dämon-Hauptprozesses

Beendigung der Login-Shell. Zur Verlagerung des Prozesses in den Hintergrund sind folgende Schritte nötig:

1. Erzeugung eines Kindprozesses mittels `fork()`
2. Beendigung des Vaterprozesses
3. Schließen der Standardausgabe und der Standardfehlerausgabe des Kindprozesses

Ein so gestartetes Programm hat keine Möglichkeit mehr interaktiv Daten auszutauschen, ist andererseits aber nicht mehr vom Vorhandensein eines Ausgabekanals abhängig. "Normale" Programme werden beendet, wenn die Standardausgabe verloren geht.

Nach dem Wechsel in den Hintergrund wird die schon beschriebene Schleife durchlaufen. Eine Änderung zum bisher beschriebenen Prinzip ist die Interpretation des `HUP`-Signals. Dieses Signal wird von vielen Dämon-Prozessen als Zeichen dafür interpretiert, daß sich etwas geändert hat[1]. Im vorliegenden Programm wird das `HUP`-Signal vom NFS-Dämon gesendet, um zu signalisieren, daß der Status einer Datei sich verändert hat. Ein Empfang des `HUP`-Signals bringt den Dämon dazu die Pause am Ende der Schleife zu beenden, um sofort auf die neue Situation reagieren zu können. Wird während eines Schleifendurchlaufs ein `HUP`-Signal empfangen, wird diese Pause übersprungen und sofort mit einem neuen Durchlauf begonnen.

Die bei der Durchsuchung der Verzeichnisse gefundenen Dateien werden nach Kategorien in Listen sortiert. Diese Listen sind mit `local` deklariert. Variablen die mit `local` deklariert sind, sind in Perl in sämtlichen aufgerufenen Unterfunktionen sichtbar. Über diese Listen erhalten die Bedingungsprüfungsfunktionen die Information über in Frage kommende Dateien.

4.1.1 Die Module

Ein wichtiger Punkt bei der Implementierung von Programmen ist die Modularisierung. Deren Aufgabe ist es die Funktionalität des Programms in unabhängige Bereiche aufzuteilen.

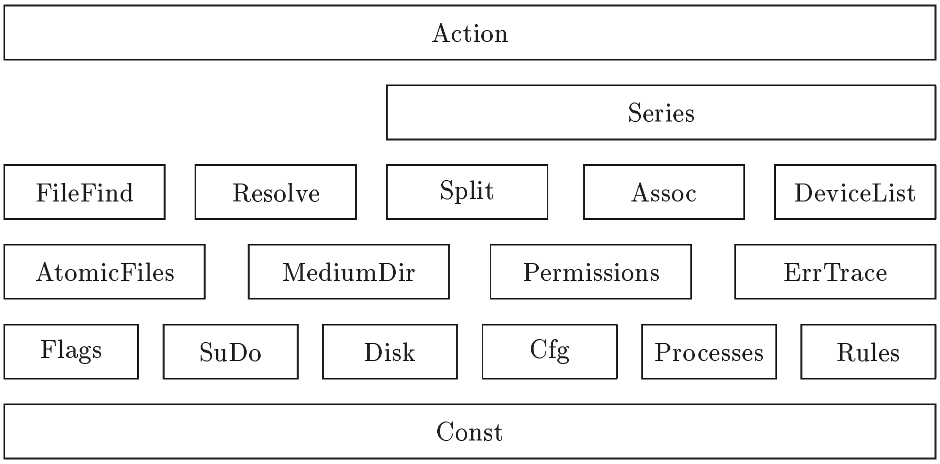
Nach [Lud] sind dabei unter anderem folgende Punkte zu berücksichtigen:

- Es sollen einerseits nicht zu große und andererseits nicht zu viele Komponenten entstehen und jede Komponente soll eine leicht erkennbare Funktion im Gesamtsystem haben.
- Es ist eine sinnvolle Gliederung zu schaffen und spezifische Teile des Systems sind zu konzentrieren.
- Es soll eine parallele Entwicklung und ein getrennter Test der Teile möglich sein.

Das Programm ist in 18 Module aufgeteilt. Diese Module sind meist sehr klein und beinhalten Funktionen in einem gemeinsamen Aufgabenbereich.

Abbildung 4.2 zeigt eine Übersicht über die wichtigsten Modulabhängigkeiten. Die Module sind in verschiedenen Schichten aufgebaut, wobei Module höherer Schichten Module niedrigerer Schichten verwenden.

[1]Der NFS-Server-Dämon liest nach Empfang von `HUP` seine Konfigurationsdateien neu ein.



(Module höherer Schichten benutzen Module niederigerer Schichten.)

Abbildung 4.2: Modul-Schichten

Im Modul `Action` sind die eigentlichen Bedingungs- und Aktionsfunktionen implementiert. Dieses Modul benutzt direkt oder indirekt sämtliche anderen Programmmodule.

Das Modul `Const` beinhaltet allgemeine Konstantendefinitionen, die von allen anderen Modulen verwendet werden.

Hier sollen nur kurz die Aufgabenfelder der Module beschrieben werden. Genaue Funktionsaufrufe und -beschreibungen sind im *POD*-Format im Quellcode[2] enthalten und über spezielle Programme, z.B. `perldoc` daraus zu extrahieren. Abbildung 4.3 zeigt ein Beispiel einer von `perldoc` erzeugten Dokumentation.

Const.pm: Diese Modul enthält lediglich Konstantendeklarationen, die vom Programm verwendet werden. Alle vom Programm verwendeten festen Zeichenketten wie z.B. die Namen der Konfigurationsparameter und die Namen der verwendeten Spezialdateien (z.B. `.assoc` und `.medium_info`) sind hier definiert. Das Modul `Const.pm` enthält keine Funktionen.

Flags.pm: Aufgabe des Moduls `Flags.pm` ist die Behandlung von Dateiattributen und Zugriffsrechten. Die enthaltenen Funktionen bilden Dateizustände (z.B. "Datei ist wiederhergestellt") auf Datei-Zugriffsrechte ab. Die aufrufenden Funktionen müssen nur die in `Const.pm` definierten Dateizustände kennen, die Umsetzung in reale Zugriffsrechte führt `Flags.pm` durch.

Cfg.pm: Das Modul `Cfg.pm` stellt Funktionen zum Auslesen von Konfigurationsdateien zur Verfügung. Konfigurationsdateien werden zum leichteren Zugriff in *hash-tables* geladen. Ein Festplattenzugriff ist darum nur einmal, beim Starten des Programms, nötig ist. Konfigurationsdateien sind nach dem Muster

```
<parameter>=<wert>
```

[2]Der Perl-Interpreter erkennt *POD*-Dokumentation im Quellcode und behandelt diese wie Kommentare.

```
Cfg(3pm)       User Contributed Perl Documentation       Cfg(3pm)


package Cfg - description
       this package contains functions to read config-files into
       a hash-table

read_cfg
       read the information of an configfile into a hash, a
       configfile contains the paramters and values in the form

       <parameter>=<value>

       whitespaces before/between/behind <parameter>, "=" and
       <value> are ignored

       call

       %cfg_hash = read_cfg ($filename);

       parameters

       $filename = name of configfile

       return

       returns the configfile-contents as a hash-table

read_cfg_structured
       read the information of an configfile into a hash, a
       configfile contains the parameters and values in the form

       <parameter>=<value>

       whitespaces before/between/behind <paramter>, "=" and
       <value> are ignored

       the read_cfg_structured-function is also able to interpret
       sections, a section is enclosed between "{" and "}" and
       needs a section-id and a section-parameter, it looks like
       this

       <section-parameter> <section-id> = {
            <parameter1>=<val1>
            <parameter2>=<val2>
       }
 ...
```

Abbildung 4.3: Auszug aus der von `perldoc` erzeugten Dokumentation zum Modul `Cfg.pm` aufgebaut. Eine Zuordnung für den Parameter `client_dir` könnte z.B. so aussehen:

```
client_dir=/fasy/clients
```

Disk.pm: Das Modul `Disk.pm` besteht aus Funktionen, die im Zusammenhang mit Zugriffen auf die Festplatte stehen. So gibt es z.B. eine Funktion zur Umwandelung eines relativen Pfadnamens in einen absoluten, oder eine Funktion zum Auslesen des Ziels eines symbolischen Links.

AtomicFiles.pm: Diese Modul implementiert die in Abschnitt 2.3.2 beschriebene Logik zur "atomaren" Erzeugung von Dateien. Die Funktionen stellen die Benutzung der Dateien weitgehend transparent dar. Beim Öffnen einer Datei wird ein Dateihandle zurückgegeben, welches wie ein ganz normales Dateihandle benutzt werden kann. Erst wenn die Datei geschlossen wird, wird sie für die anderen Programmteile sichtbar.

Assoc.pm: `Assoc.pm` stellt Funktionen zum Finden der passenden Assoziation zu einer zu registrierenden Datei zur Verfügung.

Resolve.pm: Die Verwaltung der Referenzdateien ist Aufgabe des Moduls `Resolve`. Es bietet Funktionen zur Erzeugung, zum Auslesen und Interpretieren der Referenzdateien.

Rules.pm: Das Modul `Rules.pm` enthält Funktionen zur Verwirklichung der regelbasierten Programmierung. Es wird eine Liste von Regeln (Bedingungen und Aktionen) verwaltet, wobei Bedingungen und Aktionen durch Funktionen repräsentiert werden. Beim Aufruf der Funktion `check_rules()` werden die Bedingungen der Reihe nach getestet. Gibt der Aufruf einer Bedingungsfunktion etwas anderes als eine leere Liste zurück, so wird die zugehörige Aktion mit den Rückgabewerten der Bedingungsprüfungsfunktion aufgerufen. Jede Bedingung wird so lange getestet, bis sie nicht mehr erfüllbar ist.

Action.pm: Dieses Modul besteht aus den Funktionen zur Bedingungsprüfung und Aktionsausführung.

DeviceList.pm: Mit `DeviceList` wird die Datenstruktur zur Verwaltung der benutzten Sicherungsgeräte bezeichnet. Das Modul `DeviceList.pm` enthält Funktionen zur Erzeugung und zum Zugriff auf diese Liste.

ErrTrace.pm: Dieses Modul implementiert Funktionen zur Fehlerbehandlung. Einerseits werden Fehler- und Programmablaufinformationen in eine Datei (Logdatei) geschrieben, andererseits wird der Administrator periodisch per E-Mail über den Systemzustand informiert.

Split.pm Das Modul `Split.pm` stellt Funktionen zum Teilen und Zusammensetzen von Dateien zur Verfügung.

FileFind.pm Dieses Modul enthält Funktionen zum Durchsuchen der Verzeichnisse und zur Kategorisierung der enthaltenen Dateien.

Processes.pm: Bietet Funktionen zur Synchronisation von Prozessen an, z.B. zur Prüfung, ob ein bestimmtes Gerät von einem anderen Prozeß verwendet wird.

SuDo.pm: `SuDo.pm` stellt eine Schnittstelle für Aktionen dar, die als `sudo`-Skripte abgearbeitet werden müssen. Es wird das entsprechende `sudo`-Skript aufgerufen, die Parameter werden in der Kommandozeile übergeben. Die Rückgabewerte der Funktion werden über die Standardausgabe des `sudo`-Skripts abgefangen und an die aufrufende Funktion zurückgegeben. Der Aufruf von `sudo`-Skripten findet so vollständig gekapselt statt.

Permissions.pm: Das Modul `Permissions.pm` beinhaltet Funktionen zur Prüfung der Konfigurationsberechtigungen.

Series.pm: Dieses Modul enthält Funktionen zur Bearbeitung von Serien, u.a. zur Erzeugung von Serienverzeichnissen, Feststellung des Serien-Besitzers und zur Prüfung des verfügbaren Platzes innerhalb einer Serie.

MediumDir.pm: Verwaltungsaufgaben in den Medienverzeichnissen werden vom diesem Modul gelöst. Es enthält z.B. Funktionen zur Ermittlung des freien Platzes eines Mediums usw.

4.2 SUDO-Skripte

Die Implementierung der SUDO-Skripte teilt sich in die drei bereits beschriebenen Skripte auf. Da es sich bei diesen Skripten um eigenständige Programme handelt, muß überlegt werden, auf welchem Wege die Parameter und Rückgabewerte vom Hauptprogramm in ein SUDO-Programm und zurück gelangen.

Das “Wrapper”-Skript

Zum Aufruf von Funktionen mit relativ einfachen Rückgabewerten (einer oder mehreren Zeichenketten) wird das “Wrapper”-Skript verwendet. Dem Skript wird der Name einer Funktion und deren Aufrufparameter übergeben. Die Rückgabewerte des Funktionsaufrufs werden in die Standardausgabe geschrieben und können so vom Hauptprogramm interpretiert werden.

Der Aufruf des “Wrapper”-Skripts und die Auswertung der Standardausgabe werden vom Modul `SuDo.pm` zur Verfügung gestellt. Die Funktion `do_call()` wird mit der aufzurufenden Funktion und deren Parametern aufgerufen. Zurückgegeben wird eine Liste der Rückgabewerte.

Auslesen/Schreiben von Dateien als `root`

Auslesen und Schreiben von Dateien ist etwas komplizierter, da evtl. eine große Datenmenge übertragen wird. Eine Übertragung mit den Mitteln des “Wrapper”-Skripts ist also nicht sinnvoll.

Für die Lösung dieses Problems können die unter UNIX häufig verwendeten Pipes benutzt werden:

Das Hauptprogramm startet ein SUDO-Skript, welches die gewünschte Datei öffnet. Die Daten werden über Standardaus- bzw. -eingabe übermittelt. Zu beachten ist nur, daß eine neu erzeugte Datei nach dem Schreibvorgang dem richtigen Benutzer zugeordnet wird.

Unter Perl kann die Standardein- bzw. ausgabe eines gestarteten Programms durch ein Dateihandle repräsentiert werden, so daß das Programm seine Lese- bzw. Schreiboperationen mit den üblichen Funktionen ausführen kann.

Nachteile dieser einfachen Methode sind, daß eine Datei nicht gleichzeitig zum Lesen und zum Schreiben geöffnet werden kann und daß keine Positionierung des Zeigers möglich ist. Für das vorliegende Programm reicht diese Lösung allerdings aus.

4.3 Änderungen am NFS-Dämon

Als Grundlage wird der NFS-Server von Olaf Kirch in der Version 2.2beta47 verwendet. Dieser Server implementiert NFS Version 2.

Das Hauptprogramm (in der Datei `nfsd.c`) interpretiert die Aufrufparameter, registriert den RPC-Port, wechselt in den Hintergrund und startet die RPC-Bearbeitung.

Bei einer Anfrage wird die Funktion `nfs_dispatch()` aus der Datei `nfs_dispatch.c` aufgerufen. Diese Funktion ruft, je nach Art der Anfrage, eine der NFS-Funktionen (`read()`, `write()`, `lookup()` usw.) auf und schickt die Antwort zurück.

Eine Änderung an dieser Funktion bewirkt, daß bei einer Leseanfrage zunächst die Flags der zu lesenden Datei geprüft werden. Ist die Datei nicht im Status "restauriert", so wird sie als "zu restaurieren" markiert. Dem Archivierungsdämon wird ein `HUP`-Signal gesendet.

Ist der NFS-Server im blockierenden Modus gestartet worden, wird keine Antwort zurückgeschickt, sonst wird eine Antwort mit dem Rückgabewert `NFSERR_JUKEBOX` zurücksendet.

Sämtliche Änderungen können beim Übersetzen in der Datei `config.h` durch Definition bzw. nicht Definition des Makros `FASY` ein- bzw. ausgeschaltet werden. Beim Programmaufruf kann die Funktionalität mit einem der Parameter `--fasy` oder `--fasy-block` eingeschaltet werden. Ist keiner dieser Parameter angegeben, verhält sich das Programm wie ein normaler NFS-Server.

Kapitel 5

Bewertung und Ausblick

5.1 Bewertung

FASY, das in dieser Arbeit entstandene Archivierungssystem wurde auf einem Server der Abteilung Betriebssoftware installiert und getestet. Es soll dort zur Archivierung von Sicherungen verschiedener Benutzer verwendet werden. In dieser und ähnlichen Umgebungen (ein Server mit Bandroboter, 5-10 Benutzer) ist nicht mit größeren Problemen zu rechnen.

Ein Einsatz von **FASY** ist überall dort sinnvoll, wo große Menge von Daten anfallen und die Verfügbarkeit der Daten wichtiger als die Zugriffsgeschwindigkeit ist.

Durch die freie Verfügbarkeit von **FASY** eröffnen sich auch Anwendungsbereiche, in denen sich der Einsatz von teuren Produkten wie ADSM oder HP OpenView OmniStorage nicht lohnt.

Abzuwarten bleibt, wie sich das Programm hinsichtlich Geschwindigkeit in großen Installationen mit mehreren Bandrobotern und sehr vielen Client-Rechnern verhält.

Die Akzeptanz bei Systemadministratoren hängt auch von der Verfügbarkeit eines NFS-3-Server und -Clients ab. Denn das Verhalten der vorgestellten Lösungen für NFS Version 2 ist für einfache Benutzer wahrscheinlich schwer verständlich.

Ein NFS-Server für NFS Version 3 und ein NFS-Client mit korrekter Interpretation des NFS-3-Rückgabewerts `NFS3ERR_JUKEBOX` würde hier Abhilfe schaffen.

Für den Kernel-NFSD gibt es von der Firma SGI mittlerweile einen Patch für NFS Version 3 (siehe im Internet unter [wwwo]). Dieser Server könnte nach einer Anpassung in **FASY**-Systemen zum Einsatz kommen.

5.2 Ausblick

Dieses Kapitel beschreibt Ideen zu möglichen Erweiterung von **FASY**. Einige dieser Erweiterungen sind schon sehr weit durchdacht und können vermutlich in kurzer Zeit implementiert werden. Die Realisierbarkeit anderer Ideen ist noch zu untersuchen.

5.2.1 Erweiterungen/Portierungsmöglichkeiten

Erweiterung zur Crash-Sicherheit

Eine Erweiterung des Systems zur Wiederherstellung nach einem Systemausfall würde viele neue Anwendungsmöglichkeiten bieten.

Da die transaktionsorientierte und regelbasierte Logik des Programms ohnehin sehr viele Fehler und Ausfälle toleriert, muß nur sichergestellt werden, daß sämtliche Daten auch nach einer Zerstörung der Festplatte wiederherzustellen sind.

Eine solche Erweiterung könnte wie folgt realisiert werden:

- Der Systemzustand wird regelmäßig auf einer eigenen Serie gesichert, so daß nach einem Systemzusammenbruch mit der letzten auf dieser Serie vorhandenen Datei ein relativ neuer Stand wiederhergestellt werden kann.
- Sämtliche Medien, die nach diesem Stand der Sicherung noch nicht voll belegt waren, sind einzulegen.Ebenso alle Medien, die erst nach der Sicherung des Systemzustands initialisiert wurden.
- In den gesicherten Dateien ist neben den Daten selbst auch die Referenzdatei und der Ort der Originaldatei gesichert. Mit diesen Daten können Dateien, die nach der letzten Sicherung des Systemzustands gesichert wurden, ebenfalls wiederhergestellt werden.

Mit diesem System können alle Dateien, die schon gesichert wurden inklusive ihrer Referenzdateien, wiederhergestellt werden.

Offen ist die Frage, wie die zusätzlichen Daten, die Referenzdatei und der Pfad der Originaldatei, auf dem Medium gesichert werden. Möglich wäre dies z.B. indem im Medien-Verzeichnis zu jeder Datei ein (symbolischer) Link auf die Originaldatei abgelegt wird. Beim Sichern wird dieser Link und die Referenzdatei mit der zu sichernden Datei selbst in einer TAR-Datei auf das Medium geschrieben.

Dieses System hat noch weitere Vorteile:

- Die Daten können relativ leicht auch ohne **FASY** wiederhergestellt werden (mittels `tar`).
- Es kann sehr leicht festgestellt werden, welche Dateien auf dem Band archiviert wurden.
- Teilkopien enthalten in den Referenzdateien Verweise auf die anderen Teile.

Recycling von Medien

Archivierte Daten müssen meist nur für eine bestimmte Zeit verfügbar sein. Für bestimmte Daten gibt es zum Beispiel gesetzliche Aufbewahrungfristen.

Das Archivierungssystem löscht Dateien und Verwaltungsdaten, die eine bestimmte Zeit lang nicht mehr benutzt wurden. Die Medien selbst, bzw. die Daten auf ihnen, werden aber nicht gelöscht. Denkbar wäre ein System, das Medien nach einer bestimmten Zeit für eine Wiederverwendung freigibt.

Solche Medien könnten dann ein neues Label erhalten und mit neuen Daten beschrieben werden.

"Forward-Reading"

Eine Möglichkeit die Zugriffgeschwindigkeit des Systems zu erhöhen wäre die Idee des "Forward-Readings". Wird eine Datei restauriert, besteht eine gewisse Wahrscheinlichkeit, daß ähnliche Dateien, die etwa zur gleichen Zeit gesichert wurden, ebenfalls benötigt werden.

Da ähnliche Dateien in Serien zusammengefaßt sind, befinden sich solche Dateien meist auf dem gleichen Medium. Werden statt der einen ausgewählten Datei mehrere oder alle Dateien von dem Medium wiederhergestellt, sind diese sofort verfügbar, wenn darauf zugegriffen wird.

Diese Idee könnte durch Veränderung eines bestehenden Treibers realisiert werden.

Portierung auf SMB/Samba

Das bisherige System bietet den Benutzern die Client-Verzeichnisse über einen NFS-Export an. Dies funktioniert problemlos mit UNIX und ähnlichen Systemen. Unter anderen Systemen, beispielsweise Microsoft Windows, kann auf solche Verzeichnisse standardmäßig aber nicht zugegriffen werden. Auf solchen Systemen muß zur Einbindung von NFS-Verzeichnissen eine eigene Software gekauft und installiert werden.

Microsoft Windows unterstützt standardmäßig das SMB-Protokoll zur Anbindung von Verzeichnissen fremder Rechner.

Eine Erweiterung von **FASY** dahingehend, daß auch SMB unterstützt wird, müßte ähnlich wie die Anpassung des NFS-Servers realisiert werden. Als Grundlage für ein solches Projekt kann das frei verfügbare Programm SAMBA verwendet werden.

Der Quellcode von SAMBA kann im WWW unter [wwwq] bezogen werden. Informationen zu SAMBA sind im WWW unter [wwwp] zu erhalten, Probleme des SMB-Protokolls werden in der Newsgroup [new] behandelt.

Die Installation und Konfiguration von SAMBA wird in [Car99] und [Bla98] beschrieben.

Portierung auf andere Betriebssysteme

Eine Forderung der Aufgabenstellung war eine möglichst große Portabilität.

Der verwendete NFS-Dämon ist ab der Version 2.0 nur unter Linux lauffähig, eine Anpassung des NFS-Dämons des Zielsystems ist also auf jeden Fall erforderlich. Die notwendigen Änderungen sind im Abschnitt 4.3 beschrieben.

Das Dämonprogramm ist in Perl implementiert und daher theoretisch auf allen Plattformen lauffähig, auf denen Perl verfügbar ist.

Folgende Eigenschaften von **FASY** sind bei einer Portierung zu beachten:

- Das Zielsystem muß die UNIX-spezifischen Dateiattribute wie Eigentümer und Zugriffsrechte unterstützen. Sind bestimmte Rechte nicht verfügbar, muß eventuell die Zuordnungstabelle *Zustand*→*Flag* im Modul `Flags.pm` angepaßt werden.
- Das Dateisystem sollte lange Dateinamen unterstützen, ist dies nicht der Fall, müssen die Masken für Referenzdatei etc. in der Datei `Const.pm` angepaßt werden.

- Im Gegensatz zu der Implementierung in Linux können nach [Tan95], Kapitel 7.3 in einigen UNIX-Varianten Signale nur zwischen Prozessen einer Gruppe ausgetauscht werden. Ein Prozeß kann demnach nur Signale an seine Vorfahren oder seine Kindprozesse senden. Ein NFS-Dämon, der dem **FASY**-Dämon ein Signal schicken soll, muß auf einer solchen Plattform also entweder diesen Dämon starten oder selbst von ihm gestartet worden sein.
- Der Gerätetreiber für Bandroboter verwendet die Programme `mt`, `mtx` und `tar`. Eventuell ist hier eine Anpassung erforderlich.

Treiber für weitere Geräte

FASY kann über die Erstellung von Treibern an unterschiedliche Sicherungsgeräte angepaßt werden. Die Erstellung von Gerätetreibern ist im Handbuch beschrieben. Denkbar wäre zum Beispiel eine Unterstützung von verschiedenen Bandlaufwerken, CD-Brennern, CD-Robotern, ZIP-Laufwerken usw. Auch die Implementierung eines Treibers für Diskette wäre möglich, scheint aber angesichts der geringen Kapazität nicht lohnend.

Ein Dummy-Treiber, der die Daten in bestimmten Verzeichnissen einer Festplatte ablegt, wurde zu Testzwecken implementiert. Dieser Treiber könnte z.B. für eine Auslagerung von Daten auf Wechselplatten oder langsame Festplatten benutzt werden.

Automatische Komprimierung/Verschlüsselung

Eine weitere Idee, die über spezielle Treiber realisiert werden könnte, ist eine automatische Komprimierung oder Verschlüsselung der zu sichernden Daten.

Für sicherheitsrelevante Anwendungen ist vor allem die automatische Verschlüsselung interessant. Im bisherigen System können die Daten auf den Medien von jedem wiederhergestellt werden, der die zugehörigen Medien besitzt.

Es könnten zwei Stufen der Verschlüsselung eingeführt werden:

1. Die Verschlüsselung über einen globalen Schlüssel, der nur dem Systemadministrator bekannt ist.
2. Verschlüsselung über Serien-spezifische Schlüssel, die der Serienbesitzer festlegen kann.

Der Treiber verschlüsselt die Daten bevor sie auf das Medium geschrieben werden und entschlüsselt sie automatisch beim Lesen des Mediums. Benutzer, die in Besitz des Medium sind, können die Daten nur wiederherstellen, wenn sie die nötigen Schlüssel kennen.

5.2.2 Weitere Nutzungsmöglichkeiten

Administrator-Schnittstelle über HTTP

Zur Lokalisierung von Fehlern oder zu Testzwecken ist es in einigen Fällen sinnvoll, die Arbeit eines Programms zu überprüfen. Da **FASY** sämtliche relevanten Informationen persistent auf der Platte ablegt, ist das zwar möglich, allerdings müssen dazu verschiedene Dateien in verschiedenen Verzeichnissen beobachtet werden.

Nützlich wäre ein Programm, welches relevante Informationen aufbereitet und in übersichtlicher Form anzeigt. Heute wird häufig das im World Wide Web verwendete *HTTP*-Protokoll benutzt um Informationen systemunabhängig und trotzdem optisch ansprechend anzuzeigen. Zur Anzeige veränderlicher Daten wird auf dem Server-Rechner ein CGI-Skript gestartet, welches die Informationen als HTML-Seite aufbereitet an den Browser übermittelt.

CGI-Skripte sind sehr häufig in Perl realisiert, was sich auch in diesem Fall anbietet, da die darzustellenden Informationen größtenteils aus Dateien bzw. Dateiflags gewonnen werden.

Möglich wäre z.B. eine Anzeige der folgenden Informationen:

- Status der benutzten Geräte, Kennungen der eingelegten Medien (kann aus den Gerätestatusdateien in `/tmp/fasy/tmp_dir_device_state` ausgelesen werden)
- Medien auf denen Daten gesichert werden sollen/von denen Daten wiederherzustellen sind (diese sind an den entsprechend gesetzten Flags der zugehörigen Medien-Verzeichnisse zu erkennen)
- Anzahl der Kopien einer Client-Datei und Anzeige der Medien auf denen diese gesichert sind (kann aus der Referenzdatei ausgelesen werden)
- Auswahl einer Kopie zur Wiederherstellung (Erzeugung eines symbolischen Links zum Anstoßen der Wiederherstellung)
- Konfigurationsdaten, Serien-, Assoziations-, Berechtigungsdaten

Mehrstufigkeit der Sicherung

In einem System mit unterschiedlich schnellen Datenträgern wäre auch eine mehrstufige Sicherung sinnvoll.

Daten könnten z.B. nacheinander auf folgende Medien ausgelagert werden.

- schnelle Platte
- langsame Platte
- Magnetband

Ein solches System könnte realisiert werden, indem mehrere **FASY**-Instanzen gleichzeitig laufen. Ein Treiber müßte die Dateien der ersten Instanz statt auf einem Medium in einem Client-Verzeichnis der zweiten **FASY**-Instanz sichern. Hierzu könnte z.B. der bereits implementierte Dummy-Treiber verwendet werden.

Dabei sind folgende Punkte zu beachten:

- Beide **FASY**-Instanzen sollten unter verschiedenen Benutzern laufen.
- Beide Instanzen müssen auf unterschiedliche temporäre Verzeichnisse zur Kommunikation zugreifen.
- Jede Instanz braucht ihre eigenen Client- und Medienverzeichnisse. Ein Treiber überträgt die Dateien von den Medien-Verzeichnissen der einen Instanz in die Client-Verzeichnisse der nächsten Instanz.

Geräte im Netz zur Sicherung nutzen

Diese Idee begründet sich auf der Tatsache, daß die Zahl der Geräte in jedem Rechner begrenzt ist. Sollen zur Archivierung mehrere Geräte angezogen werden, soll es auch möglich sein, Geräte anderer Rechner zur Archivierung zu nutzen.

Nach einer Erweiterung des Abteilungsservers *linde* der Abteilung Betriebssoftware, hat sich genau diese Situation ergeben: nach dem Einbau neuer Festplatten war kein Platz mehr für den Bandroboter. Aus dieser Situation heraus entstand ein Treiber für eine Sicherung über das Netzwerk, eine Anpassung des Dämon-Programms ist nicht notwendig.

Der Rechner, auf dem **FASY** läuft, wird im Folgenden **FASY-Server** genannt, der Rechner der ein Sicherungsgerät zur Verfügung stellt wird **Geräteserver** genannt.

Eine Implementierung eines solchen Treibers ist relativ einfach.

- Sämtliche Geräteaktionen müssen über ein remote-shell-Programm (z.B. `rsh` oder `ssh`) an den Geräteserver weitergegeben werden.
- Alternativ kann für Bandlaufwerke auch das Programm `rmt` zum Zugriff auf entferne Laufwerke benutzt werden.

Die Implementierung des Netzwerk-Gerätetreibers für Bandroboter ist im **FASY**-System enthalten.

5.2.3 Performancesteigerung durch compilierten Code

In der verwendeten Perl-Version 5.005 befindet sich der Perl-Compiler noch in einem experimentellem Stadium. Hat dieser Compiler einmal einen stabilen Zustand erreicht ist es denkbar auch das **FASY**-System damit zu übersetzen.

Eine Übersetzung wird sicher an einigen Stellen die Geschwindigkeit des Programms steigern, im großen und ganzen wird sich an der Performance des Programms allerdings nicht viel ändern. Ein Großteil der Funktionen greift auf die Festplatte zu. Da diese Festplattenzugriffe durch die Geschwindigkeit der Festplatte bestimmt werden ist zu erwarten, daß sich an der Geschwindigkeit des Programms auch nach einer Übersetzung in Binärcode nichts ändern wird.

Literaturverzeichnis

[Bec99] Michael Beck. *Linux Kernelprogrammierung*. Addison-Wesley, 5. edition, 1999. ISBN 3-8273-1476.

[Bla98] John D. Blair. *SAMBA - Integrating UNIX and Windows*. Specialized Systems Consultants, Inc. (SSC), 1998. ISBN 1-57831-006-7.

[BWK88] Dennis M. Ritchie Brian W. Kernighan. *the C programming language*. Pretince-Hall International, 1988. ISBN 0-13-110362-8.

[Cal95] B. Callagham. RFC 1813. Technical report, Sun Microsystems, Inc., 1995. http://www.cis.ohio-state.edu/htbin/rfc/rfc1813.html.

[Car99] Gerald Carter. *Teach Yourself Samba in 24 Hours*. Sams, 1999. ISBN 0-672-31609-9.

[Chr99] Tom Christiansen. *Perl Kochbuch*. O'Reilly, 1999.

[dS] James da Silva. *AMANDA manpage*. ftp://ftp.amanda.org/pub/amanda.

[Fen98] Kevin Fenzi. *Linux Security HOWTO, V0.9.11*, 1998. ftp://sunsite.unc.edu/pub/Linux/docs/HOWTO.

[Fri95] AEleen Frisch. *Essential system Administration*. O'Reilly & Associates Inc., 2nd edition, 1995. ISBN 1-56592-127-5.

[Gra97] Mark Grand. *JAVA Fundamental Classes Refernce*. O'Reilly & Associates, Inc., 1997. ISBN 1-56592-241-7.

[Gro89] Network Working Group. RFC 1813. Technical report, Sun Microsystems, Inc., 1989. http://www.cis.ohio-state.edu/htbin/rfc/rfc1094.html.

[Gru] Dick Grund. *cvs manpage*. http://www.cyclic.com, http://www.loria.fr/ molli/cvs-index.html.

[Haj98] Farid Hajji. *Perl - Einführung, Anwendung, Referenz*. Addison-Wesley, Bonn, 1998.

[Kir] Olaf Kirch. Linux NFS-Server. ftp://linux.mathematik.tu-darmstadt.de/pub/linux/people/okir.

[Kno97] Fank Knobloch. *Java*. bhv Verlags GmbH, 1997. ISBN 3-89360-335-2.

[Kof98] Michael Kofler. *Linux Installation, Konfiguration, Anwendung*. Addison-Wesley Longman Verlag, 3. edition, 1998. ISBN 3-8273-1304-X.

[Lub96] Holger Lubitz. *Perl 5 - Schnellübersicht*. O'Reilly, 1996.

[Lud] Prof. Jochen Ludewig. Skript Software für die Anwendung, Univ. Stuttgart, Institut für Informatik, WS 1994/95.

[Nem89] Evi Nemeth. *UNIX System Administration Handbook*. Pretince Hall Software Series, 1989. ISBN 0-13-933441-6.

[new] Newsgroup zum SMB-Protokoll. comp.protocols.org.

[NP99] Clay Irving Nate Patwardhan. *Programmieren mit Perl Modulen*. O'Reilly, 1999.

[Pee93] Jerry Peek. *Unix Power Tools*. O'Reilly & Associates Inc., 1993. ISBN 0-679-79073-X.

[Sed92] Robert Sedgewick. *Algorithmen in C*. Addison-Wesley, 1992. ISBN 3-89319-669-2.

[Sri98] Sriram Srinivasan. *Fortgeschrittene Perl Programmierung*. O'Reilly, 1998.

[Tan95] Andrew S. Tanenbaum. *Moderne Betriebssysteme*. Hanser Studienbücher der Informatik, 2. edition, 1995. ISBN 3-446-18402-3.

[Tea98] The Amanda Development Team. *AMANDA-README*, 1998. ftp://ftp.amanda.org/pub/amanda.

[Tor] Linus Torvalds. Linux Kernel. ftp://ftp.kernel.org/pub/linux/kernel/v2.2/.

[Vro99] Johan Vromans. *Perl 5 kurz & gut*. O'Reilly Verlag, Köln, 1999.

[Wal90] Larry Wall. *Programming Perl*. O'Reilly & Asociates, 1990.

[Wal97] Larry Wall. *Programmieren mit Perl*. O'Reilly, 1997.

[Wal99] Larry Wall. *Perl Programmers Reference Guide*, July 1999. perl 5.005 patch 03.

[wwwa] Tivoli Homepage. http://www.tivoli.com.

[wwwb] IBM-Seite zu ADSM. http://www.de.ibm.com/service/rs6000/ImpADSMSAP.html.

[wwwc] ADSM Anleitung der Uni Stuttgart. http://www.uni-stuttgart.de/rus/Bi/BI_html/BI98/1+2/file5.html.

[wwwd] ADSM Anleitung der TU München. http://www.lrz-muenchen.de/services/datenhaltung/adsm/einf/.

[wwwe] ADSM Anleitung der Uni Heidelberg. http://aixmita1.urz.uni-heidelberg.de/UnixCluster/Hinweise/Hilfe/System/Adsm/intro1.html.

[wwwf] ADSM Anleitung der Uni Hohenheim. http://www.uni-hohenheim.de/rz/sys/adsm/wasistADSM.htm.

[wwwg] HP-OpenView Produktinformation. http://www.hpov.de/products.shtml.

[wwwh] Hewlett Packard - Homepage. http://www.hp-online.com.

[wwwi] Computer Associates, Inc. - Homepage. http://www.cai.com.

[wwwj] LSC Inc. - Homepage. http://www.lsci.com.

[wwwk] Veritas - Homepage. http://www.veritas.com.

[wwwl] Tracer Technologies - Homepage. http://www.tracertech.com.

[wwwm] The KNFSD-Homepage. http://www.CSUA.Berkeley.EDU/ gam3/knfsd/.

[wwwn] SGI - homepage zu XFS. http://oss.sgi.com/projects/xfs.

[wwwo] SGI - Homepage zu NFS Version 3. http://oss.sgi.com/projects/nfs3.

[wwwp] SAMBA - Homepage. http://www.samba.org.

[wwwq] SAMBA - Sourcecode CVS-Archiv. http://cvs.samba.org/cvs.html.

Appendix A

the fasy manpages

A.1 FASY.PL

NAME

FASY — File Archiving SYstem

fasy.pl — the FASY-daemon

SYNOPSIS

fasy.pl `[-d=`<directory`]` `[-f=`<filename>`]` `[-F]`

DESCRIPTION

FASY is a system for transparent storing and restoring files. All files, put into a certain client directory are automatically moved to a storage-medium like magnetic tape. When accessing the file (over a modified nfsd) it is automatically recovered.

fasy.pl. the FASY-daemon-process, controlls the storing and restoring of files. It should be started by the fasy-user, maybe at boottime. For configuration details see *fasy_cfg*(5). Exporting and mountuing a FASY-filesystems is explained in *fasy_nfs*(8).

This Version has currently only been tested on linux systems, namely SuSE-Linux 6.2/6.3 and Red Hat Linux 6.1. It may work in other environments.

OPTIONS

`--`**configdir**=*directory* or **-d**=*directory*

directory spezifies the directory where the configuration file *fasy.cfg* and the media configuration files, are located. The default configuration-directory is */etc/fasy*

`--`**configfile**=*filename* or **-f**=*filename*

filename gives the name and path to a different main configuration file, default is *fasy.cfg* in the configuration-directory

`--`**foreground** or **-F**

the program ist started in foreground instead of starting it as daemon process

FILES

/etc/fasy/fasy.cfg — this file includes the FASY-configuration, see *fasy_cfg*(5)

/etc/fasy/medium.∗ — this files include the FASY-medium-configuration, see *fasy_media*(5)

FASY also uses the files **.perm**, **.assoc** and **.series** in the client-directories, see *fasy_perm*(5), *fasy_assoc*(5), *fasy_series*(5).

ENVIRONMENT

FASY needs the following applications to be installed:

- **Perl 5.004 or higher (see *http://www.perl.com*)**
- **sudo**
- **any mail-program like mail**
- **TCP/IP and RPC**
- **tar and mt if using magnetic-tapes**
- **mtx if using tape-roboters (see *http://www.dandelion.com/Linux*)**

DIAGNOSTICS

(F) — fatal error, (N) — nonfatal error, (I) — information

configfile $filename does not exist or is empty!

(F) The configfile could not be found or was empty. Create the configfile or specify the path of an existing configfile using the *-f* parameter

configuration-error!!! — see logfile

(F) Error while reading the configuration file. See the logfile for further information. The logfile is configured in *fasy.cfg*, default: fasy.log in the current directory.

archive-user $uid doesn´t exist !

(F) The configured uid for the archive-user doesn´t exist. Change the configfile-entry or create the user.

sorry, you (uid=$uid) are not the archive-user, only $uid is allowed to start this program!

(F) The user who tried to start fasy is not allowed to do this. Switch to the shown user or change the configfile-entry.

Unknown option: $option

(N) The passed command line option is unknown, it is ignored.

the fasy-daemon is started

(I) the fasy daemon is started and running

EXAMPLES

```
fasy.pl -F
```

```
fasy.pl -d /etc -f /etc/fasysample.cfg
```

BUGS

The |-Operator in association-files doesn't work right now. Currently it is implemented as a synonym for the `,`-Operator.

AUTHOR

Jörg Matysiak <joerg.matysiak@gmx.de>

SEE ALSO

fasy_cfg(5), *fasy_assoc*(5), *fasy_series*(5), *fasy_perm*(5), *fasy_media*(5), *fasy_nfs*(8), *nfs*(5), *nfsd*(8), *exports*(5)

A.2 FASY.CFG

NAME

fasy.cfg — the FASY configuration file

SYNOPSIS

/etc/fasy/exports

DESCRIPTION

This file contains the main-configuration for the FASY-daemon-program.

The syntax to configure parameters is the following:

`option = value`

Empty lines are igneored. Comments start with a leading `#`.

OPTIONS

- **archive_user: Specifies the username of the FASY-user. That's the user who is allowed to start the FASY daemon-program. This user also owns the registered files and series-directories. Standard:** ***archive_user=fasy***
- **driver_path: Specifies the directory containing the subdirectory-structure with the driver-scripts. Standard:** ***driver_path=~/bin/Driver***
- **error_logfile: Gives the name of the file where errors and trace-informations should be saved. Standard:** ***error_logfile=~/fasy.log***
- **error_max_logfile_size: Sets the maximum size of a logfile in byte. If it has reached this size, the file is overwritten. Standard:** ***error_max_logfile_size=100000***
- **error_mail_cmd: Specifies the program (including parameters) which is called for sending an e-mail to the systemadministrator. The program should read the mail from STDIN. Standard:** ***error_mail_cmd=mail -s Archive-System-Status root***
- **error_mailing_interval: Sets the sequence for sending status-messages to the administrator (in seconds). Standard:** ***error_mailing_interval=1800***
- **error_to_log: Specifies the error-classes written into the logfile, known classes are:**
  fatal — fatal errors, that cause the program to stop

  nonfatal — errors/problems that don't cause the program to stop, but they should be reported to the administrator

  info — informations about the system-status

  trace — informations for degbugging etc.

  Standard: *error_to_log=info, nonfatal, fatal*

- **error_to_mail: Specifies the error-classes, which are send to the administrator. Available classes see *error_to_log*, standard: *error_to_mail=nonfatal, fatal***
- **temp_dir: Specifies the directory, into which temporary files (e.g. for communications between its sub-programs) are stored. The standard-value is *temp_dir=/tmp/fasy***
- **client_dir: Specifies the directory into which the client-users place their files to archive. Standard *client_dir=~/clients***
- **admin_dir: Specifies the directory where the administration information (e.g. the resolve-files) are stored. Usuablly this would be the client directory. Standard: *admin_dir=~/clients***
- **series_dir: Specifies the directory where the subdirectories for series and (as their subdirectories) the directories for media are created. To get best performance place this directory on the same partition like the client directory. Standard: *series_dir=~/series***
- **max_waiting_time: Specifies the maximum interval (in seconds), the daemon-program should sleep between two loop-runs. If the daemon receives a HUP-signal, it awakes and performs the next loop. Standard: *max_waiting_time=120***
- **storing_timespaces: Specifies the times when fasy is allowed to store data. This could be for example in the night when the system-load ist low. Each timespace is a set of beginning and ending-time, with a dash in the middle (HH:MM-HH:MM), each time is a set of hours and minutes, separated with a colon (HH:MM). Different timespaces can be separated by commas. Standard: *storing_timespaces=00:00–24:00***
- **file_completion_time: Specifies the time (in seconds) which a file has to be unchanged before registering it. This is needed because a file-transfer e.g. over ftp takes some time and the system should not register a file before it is completely saved. Standard: *file_completion_time=300***
- **file_wait_before_storing_time: Specifies the time (in seconds), which a file has to be in a medium-dir, before it can be stored. Standard: *file_wait_before_storing_time=200***
- **file_sparsing_time: Specifies the time (in seconds) a restored file has to be untouched before it is sparsed again. Standard: *file_sparsing_time=86400***
- **file_deletion_time: Specifies the time (in seconds) after which a sparse-file and it's corresponding resolve-file are completely removed from the system. If given a negativ value, the files are never removed automatically, standard is *file_deletion_time=-1*.**

Device-Configuration

The configuration of devices differs from the configuration of the other parameters. A device is specified by a group of parameters, which are enclosed by '{' and '}\. A device-configuration starts with the word `device` and a device-identifier-string. Between the brackets the device-parameters are specified. It has the following syntax:

```
device <device_id> = {
```

```
        slotnumber = <number of slots>
        devicetype = <device type>

        subdevice <subdevice_id> = {
                readable = <readable device-types>
                writeable = <writeable device-types>
                device = <device-file>
        }
        location =<location>
}
```

Explanation of the parameters:

- **slotnumber: Number of slots of this device.**
- **devicetype: Type of the device. Name of device-driver which is called for accessing the device.**
- **location: Specifies the location of the device. Devices can have one or more locations, separated by commas.**
- **subdevice: Specifies the parameters of any reading-/writing-unit of this device. The following parameters can sbe specified on each subdevice:**
  readable: Device types which can be read on this subdevice, separated by commas.

  writeable: Device types which can be written on this subdevice, separated by commas.

  device: The device-id. This file is interpreted by the device driver. Usually this would be the unix device file, beginning with a leading `/dev/`.

DEVICE DRIVERS

The current Version of FASY includes the following device drivers

- **Tape: The driver for single tape drives controlled by *mt*. It automatically recognizes the media types *dds1*, *dds2* and *dds3*, all other media types are realized as *tape*.**
- **MultiTapeNet: This driver supports tape drives that can hold more than one tape at the same time and can be controlled over the *mt* and *mtx* tools. The devices can be installed at the machine FASY runs on, but also on a device-server in the net. This driver automatically recognizes the media types *dds1*, *dds2* and *dds3*.**
- **DummyDriver: A dummy driver, which just copies the files into a harddisk-directory. This drivers return the media type *disk*.**

MEDIA TYPES

The following media types are supported (bracketed the drivers which support this type):

- **dds1 (multi_tape, multi_tape_net)**
- **dds2 (multi_tape, multi_tape_net)**

- **dds3 (multi_tape, multi_tape_net)**
- **disk (dummy)**

EXAMPLES

This is an exmaple of a FASY-configuration-file:

```
#
# This is the standard-fasy-configuration file
# for further information see man fasy.cfg(8)
#

#
# set the loginame of the user who is allowed to
# start the fasy-daemon and who owns the registered files
#
archive_user=fasy

#
# set the directory which includes the driver-paths
#

driver_path=~/bin/Driver

#
# trace-definitions
# possible error-classes are: fatal, nonfatal, info, trace

# the file where problems are logged
error_logfile=~/fasy.log

# the maximum-logfile-size
error_max_logfile_size=100000

# the command fo sending e-mails to the administrator
error_mail_cmd=mail -s Archive-System-Status root

# the interval for sending mails to the administrator
error_mailing_interval=1800

# the error-classes to write to the logfile
error_to_log=info, nonfatal, fatal

#the error-classes to send to the administrator
error_to_mail= nonfatal, fatal

# the temporary directory
temp_dir=/tmp/fasy

# the client-top-directory
client_dir=~/clients

# the admin-top-directory
admin_dir=~/clients

# the series-top-directory
series_dir=~/series
```

```
# the sleep-time between two loops
max_waiting_time=120

# the timespaces where storing is allowed
storing_timespaces=0:00-24:00

# the interval (in seconds), a file has to be untouched before registering it
file_completion_time=300

# the interval (in seconds), a file has to be untouched before storing it
file_wait_before_storing_time=200

# the interval (in seconds), a file has to be untouched before sparsing it
# one day=86400 secounds
file_sparsing_time=86400

# the interval (in seconds), a sparse file has to be untouched
# before it's deleted (negativ value means "never")
file_deletion_time=-1

#
# the device-definition
#

#device dummy = {
#       slotnumber = 5
#       devicetype=dummy
#       location=AT_HOME
#
#       subdevice 0 = {
#               readable=disk
#               writeable=disk
#               device=/home/fasy/dummy/slots
#       }
#}

device taperobot = {

        slotnumber=6
        devicetype=multi_tape
        location=AT_HOME

        subdevice 0 = {
                readable=dds1, dds2, dds3
                writeable=dds1, dds2, dds3
                device=/dev/nst0
        }
}
```

AUTHOR

Joerg Matysiak <joerg.matysiak@gmx.de>

SEE ALSO

fasy (8), *fasy_assoc*(5), *fasy_series*(5), *fasy_perm*(5), *fasy_media*(5), *fasy_nfs*(8), *nfs*(5), *nfsd*(8), *exports*(5)

A.3 FASY_MEDIA

NAME

medium.∗ — the FASY media-configuration-files

SYNOPSIS

/etc/fasy/medium.∗

DESCRIPTION

The media-configuration file are setting the device-type specific parameters, such as the medium-capacity. FASY examinates the configuration-directory (default */etc/fasy/*) for files with leading *medium.*. Each of this files can contain one ore more media-type definitions.

FILE FORMAT

A media-type configuration looks like this:

```
 medium <medium-type> = {
        capacity = <medium_capacity>
        min_restsize = <min_restsize>
}
```

Only supported media types can be used. For a list of supported media types see fasy.*cfg*(5).

- **medium_capacity: Sets the capacity of the specified medium-type in bytes.**
- **min_restsize: Specifies the size (in bytes) that has to be left on a medium. If there is less than *min_restsize* bytes free on a medium, it is marked as full.**

Empty lines are igneored. Comments start with a leading #.

DEVICE DRIVERS

For a list of supported device-types see fasy.*cfg*(5).

MEDIA TYPES

For a list of supported media-types see fasy.*cfg*(5).

EXAMPLES

This is an example of a FASY-medium-configuration-file for dds3:

```
# this is a sample medium-file for DDS3-tapes
# don't trust the numbers, they're just samples

medium dds3 = {
      capacity = 1200000000
      min_restsize= 100000
}
```

AUTHOR

Joerg Matysiak <joerg.matysiak@gmx.de>

SEE ALSO

fasy(8), fasy.*cfg*(5), *fasy_assoc*(5), *fasy_series*(5), *fasy_perm*(5), *fasy_nfs*(8), *nfs*(5), *nfsd*(8), *exports*(5)

A.4 FASY_PERM

NAME

.perm — the FASY permission-definition-files

SYNOPSIS

.perm

DESCRIPTION

This file specifies which users are allowed to configure which parameter.

For each permission you can define which users are explictely allowed to do this action and which users are explicitely forbidden.

The client-top-directory and all its subdirectories can contain perm files. They are interpreted top-down (from the top-directory to the subdirectories).

Each user who owns a permission is allowed to restrict this permission by creating a own **perm**-file in each sub-directory. A user who loosed any permission in a directory will not get this permission in any subdirectory of this directory, even if he is listed in the allowed-statement.

FILE FORMAT

Each line is built-up like follows:

```
<right_id>_[allowed|forbidden]=<user>[[, <user>] ...]
```

The string **all**, if given as user-id, has a special meaning. If it appears in the allowed-statement, it means: 'all users who own the permission so far will keep it'. If **all** appears in a forbidden-statement, this means: 'no user will have that permission, even if the allowed-statement contains some users'.

Possible permission-ids are:

- **assoc: The right to define file-to-series-association**
- **series: The right to define own series**

Empty lines are igneored. Comments start with a leading `#`.

EXAMPLE

```
# this will deny the guest user of defining associations,
# all other users are allowed (if they are allowed so far)
assoc_allowed = all
assoc_forbidden = guest
```

```
# this will allow series-definitions only for the users
# fasy, root and sys, all other users are forbidden
series_allowed = fasy, root, sys
# the following line is not really needed, because each
# user who is not explicitly allowed is forbidden
series_forbidden = guest
```

AUTHOR

Joerg Matysiak <joerg.matysiak@gmx.de>

SEE ALSO

fasy(8), fasy.*cfg*(5), *fasy_media*(5), *fasy_assoc*(5), *fasy_series*(5), *fasy_nfs*(8), *nfs*(5), *nfsd*(8), *exports*(5)

A.5 FASY_ASSOC

NAME

.assoc — the FASY user-association-files

SYNOPSIS

.assoc

DESCRIPTION

Each client subdirectory may contain a **.assoc**-file, which specifies which file-types will be stored on which series.

A assoc-files is interpreted only if the owner is allowed to make association-configuration, see *fasy_perm*(5) for further details on permissions.

The association file associates a list of files with a series-term. The file list can be given as shell-glob-patterns including `*` and `?`. If more than one file-descriptions do match, the first found is used. The assoc-files are read in bottom-up order:

- **First the .assoc-file in the files directory is read. If no match was found, the .assoc-file in the parent-directory is read. Then the file in the parents parent-directory and so on. Files whose owners don't have the permissions for making association configuration are ignored.**
- **The files theirselves are read top down, the first matching term is used.**

FILE FORMAT

Each line contains one rule. A rule consists out of two parts: a list of file-globs and a series-term. Both parts are separated by a colon (and a whitespace before and behind the colon).

The file-globs

The file-globs are separated with whitespaces. Therefor a file-glob **must not contain whitespaces** itself! The wildcards $*$ and *?* are allowed.

The series-term

There are the following possibilities for defining series-terms:

- **a series-name**
- **multiple series-terms, concatenated with a `','`-Operator**
- **multiple series-terms, concatenated with a `'|'`-Operator**
- **multiple series-terms, concatenated with a `'+'`-Operator**

The HARD-OR-operator `','`: Save on the first (left) given serie, even if there are some inserted media to label. If there is not enough space on the first serie and there are no more empty mediums to label for this serie, use the secound (right) serie.

The SOFT-OR-operator `'|'`: Save on the first (left) given serien if this is possible without labeling any new media. If this is impossible, then store on the second (right) given serie.

The AND-operator `'+'`: Store the file on both series, the first (left) and the second (right).

The operator-binding is stronger from top to down. The operators are interpreted from left to right.

EXAMPLES

```
*.tgz *.tar.gz : TGZ_SERIE, GZ_SERIE + TAR_SERIE, ALL_SERIE
*.jpg pic* : JPG_SERIE, PIC_SERIE | PICTURES1 + PICTURES2
* : DEFAULT_SERIE
```

Explanation:

- **first line: This rule matches for all files which names are ending with `.tgz` or `.tar.gz`. If possible those files are saved on the serie `TGZ_SERIE`, even if a new medium has to be labeld. If this serie has not enough space and there are no more empty media left to label (for this serie), then store on both series `GZ_SERIE` and `TAR_SERIE`. If there is not enough space available for one of this series then try storing on the serie `ALL_SERIE`.**
- **secound line: This rule matches for all files whichs names are ending with `.jpg` or beginning with `pic`. If possible store that files on the serie `JPG_SERIE`, even if new media has to be labeld. If there is not enough space on this serie, try the serie `PIC_SERIE`. If the file can be stored on the `PIC_SERIE` without labeling new media, store on this serie, otherwise store on the series `PICTURES1` and `PICTURES2`. (Store each file on both!)**

BUGS

The SOFT-OR operator `'|'` is currently implemented as a synonym of the HARD-OR operator `','`.

RESTRICTIONS

If you define a association, in which a file is associated twice to the same serie (e.g. `SERIE_X + SERIE_X`) you **must not** expect, that the file is stored on different media. (It may be stored two times on the same medium.)

NOTES

The administrator is responsible that there can be found any association for every file in the client-directory. This can be realized with a association-file in the highest client-directory, which contains a association for all files, represented through an `*`.

example finite association in the clients-top-dir

```
* : DEFAULT_SERIE
```

AUTHOR

Joerg Matysiak <joerg.matysiak@gmx.de>

SEE ALSO

fasy(8), fasy.*cfg*(5), *fasy_media*(5), *fasy_series*(5), *fasy_perm*(5), *fasy_nfs*(8), *nfs*(5), *nfsd*(8), *exports*(5)

A.6 FASY_SERIES

NAME

.series — the FASY series-definition-files

SYNOPSIS

.series

DESCRIPTION

Series are created by defining their parameters in a **series**-file. A defined serie can be used in association-files (see *fasy_assoc*(5)) which are located in the same directory like the **series**-file or in any subdirectory of it.

The owner of the **series**-file is the owner of all series defined in that file. The full series name is a concatenation of the series-owners loginame and the series-identification.

FILE FORMAT

A **series**-file can include one or more series-definitions, each series definition starts with the word `serie` followed by an identifier. The series parameters are enclosed in a bracketed section. For series-definition syntax see below:

```
serie <serie_id> = {
      media_types = <media_types>
      location = <location>
      write_only_when_full = <TRUE or FALSE>
}
```

The parameters are:

- **media_types: Specifies a list of media-types used for storing this serie.**
- **location: Gives one or more locations where the media for this serie can be labeld. This locations must match with at least one of the locations defined in the device-configuration (see fasy.cfg(5)), otherwise this serie will not be able to label the media.**
- **write_only_when_full: This parameter specifies the writing logic. If set to TRUE the medium will only be written if the medium-directory includes all files that will ever be stored on this medium. If set to any other value, files can be stored to the medium before the medium-directory is complete.**

Empty lines are igneored. Comments start with a leading `#`.

MEDIA TYPES

For a list of supported media-types see fasy.*cfg*(5).

EXAMPLE

```
# exmaple serie for storing TAR-files at the IFI
serie MY_TAPE_SERIE = {
       media_types = dds2, dds3
       location = UNI_STUTTGART_IFI
       write_only_when_full = FALSE
}

# serie definition example to test the dummy-driver

serie MY_DISK_SERIE = {
       media_types = disk
       location = UNI_STUTTGART_IFI, AT_HOME
       write_only_when_full = TRUE
}
```

AUTHOR

Joerg Matysiak <joerg.matysiak@gmx.de>

SEE ALSO

fasy(8), fasy.*cfg*(5), *fasy_media*(5), *fasy_assoc*(5), *fasy_perm*(5), *fasy_nfs*(8), *nfs*(5), *nfsd*(8), *exports*(5)

A.7 FASY_NFS

NAME

FASY-NFS — starting the FASY-nfsserver and mounting FASY-filesystems

SYNOPSIS

rpc.nfsd [NFSD-OPTIONS] `[--fasy]` `[--fasy-block]`

DESCRIPTION

The FASY-nfsserver exports the fasy-client-directories for transparent file-access. Accessing a stored file, which is not present locally, will start the restore-process. Depending on the option passed to the nfsserver at starttime, the accessing application will be blocked until the file is available or will return an i/o-error if the file is not available.

OPTIONS

For NFSD-OPTIONS see *nfsd*(8).

The FASY-specific options are

- **--fasy: If this option is specified the FASY-extensions will be used in nonblocking-mode. Any application which accesses a stored file gets an i/o-error, if this file is not available. Restoring of this file is started.**
- **--fasy-block: If this option is specified the FASY-extensions will be used in blocking-mode. Any application which accesses a stored file gets blocked until the file is restored.**

EXAMPLE

starting the mountserver on a specific port

```
rpc.mount -P 1002 -f ~fasy/exports

rpc.mount -P 1002 -f /etc/fasy/exports
```

starting the nfsserver on a specific port

```
rpc.nfsd -P 1001 -f ~fasy/exports --fasy

rpc.nfsd -P 1001 -f /etc/fasy/exports --fasy-block
```

mounting a fasy-filesystem on specific ports

```
mount -oport=1001 -omountport=1002 lukas:/home/fasy/clients /clients
```

/etc/fstab-entry for mounting a fasy-filesystem on specific ports

```
lukas:/home/fasy/clients /clients nfs port=1001,mountport=1002 0 0
```

AUTHOR

Joerg Matysiak <joerg.matysiak@gmx.de>

SEE ALSO

fasy(8), fasy.*cfg*(5), *fasy_media*(5), *fasy_assoc*(5), *fasy_perm*(5), *fasy_series*(5), *nfs*(5), *nfsd*(8), *exports*(5)

Anhang B

Ursprüngliche Aufgabenstellung

Diplomarbeit

Backup-und Archivierungssystem mit Medienverwaltung

Aufgabe ist es, ein Backup- und Archivierungs-System zu entwickeln, das große Datenmengen auf unterschiedlichen Medien (Magnetbänder, CD-R, etc.) sichert. Die Sicherung soll für die Benutzer möglichst transparent erfolgen.

Es existieren eine oder mehrere Holding-Disks, die vom Backup-System überwacht werden. Auf diesen Platten gibt es für jeden Client ein Verzeichnis (z.B. `/clients/<clientname>`), in das die zu sichernden Dateien abgelegt werden. Dateien, die auf diese Weise abgelegt und eine konfigurierbare Zeit lang nicht mehr verändert wurden, werden vom Backup-System mit einem Schreibschutz versehen und einem Medium zugeordnet.

Die so markierten Dateien werden zu regelmäßigen (konfigurierbaren) Zeitpunkten oder nach Ablauf von Zeiträumen auf das tatsächliche Medium gesichert. Dabei muß berücksichtigt werden, daß evtl. Restkapazitäten von Medien in einem zukünftigen Sicherungsdurchlauf noch aufgefüllt werden können.

Die Sicherung der Dateien bzw. der Zugriff erfolgt in mehreren Stufen:

a) Dateien sind im Direktzugriff des Clients (im Client-Verzeichnis)

b) Dateien sind im Direktzugriff des Servers (im Medienverzeichnis)

c) Dateien sind ausgelagert auf ein (langsames) Medium, Zugriff auf das Medium ist aber noch ohne Eingriff eines Benutzers/Administrators möglich (d.h. das Einlegen von Bändern, CDs etc. ist **nicht** nötig)

d) Dateien sind auf Sicherungsmedium ausgelagert, Eingriff des Benutzers/Administrators ist nötig, die Verwaltungsinformation (Dateinamen, -größe, -lage) ist aber noch auf dem Server vorhanden.

e) Dateien und Verwaltungsinformation sind auf Sicherungsmedien gesichert

Die Stufen b-d sollen für den Benutzer so weit wie möglich transparent gehalten werden. Im Idealfall bemerkt der Benutzer den Unterschied nur an unterschiedlichen Zugriffszeiten auf die Daten.

In Stufe e wird nicht mehr mit einem Zugriff auf die Daten gerechnet. Soll jetzt noch auf die Daten zugegriffen werden, kann das für den Benutzer nicht mehr transparent erfolgen. (Die Datei muß z.B. über ein spezielles Programm manuell rekonstruiert werden.)

Für kritische Daten soll es zusätzlich die Möglichkeit geben, daß diese auf mehrere Medien, ggf. unterschiedlichen Medien-Typs, gesichert werden. Zur Wiederherstellung der Datei muß eine einzelne Sicherungskopie ausreichen.

Die Sicherungsstrategien (z.B. Welche Datei wird auf welchem Medium gesichert?, Welche Datei verbleibt wie lange in welcher Stufe?, Welche Datei wird mehrfach gesichert?) sollen konfigurierbar gestaltet werden (Wildcards o.ä.), und zwar durch jeden Client selbst, z.B. durch eine Konfigurationsdatei im Client-Verzeichnis, in der Abweichungen von der globalen Konfiguration angegeben werden können.

Es ist auch der Fall zu berücksichtigen, daß der Server-Rechner bzw. dessen Platte ausfällt. In diesem Fall muß es möglich sein einen möglichst aktuellen und konsistenten Systemzustand aus den Backups manuell zu rekonstruieren.

Die Verwaltung der Speichermedien soll vom Backup-System möglichst transparent gehandhabt werden, eingelegte Medien sollten automatisch erkannt werden. Die Kommunikation mit dem Operator soll über EMail-Meldungen erfolgen, wobei die Texte der noch ausstehenden Aktionen zur Sicherheit in Dateien hinterlegt werden sollen.

Die Archivierungsfunktionalität wird in einem eigenen Prozeß angesiedelt, mit dem über die Attribute und Modifikationszeiten von Dateien kommuniziert werden kann. Diese untere Schicht braucht einige Funktionalitäten wie das Splitten von über mehrere Medien verteilten Dateien oder mehrfach gesicherten Dateien nicht vollständig transparent auszuführen, sondern kann hierfür z.B. zusätzliche Dateinamens-Suffixe einführen. Daneben wird ein modifizierter NFS-Daemon für den Zugriff auf die Client-Verzeichnisse bereitgestellt, der die Zugriffe transparent und ohne Umweg über Attribute ermöglicht. Die Kommunikation zwischen beiden Schichten soll möglichst einfach sein und über das Dateisystem erfolgen, so daß ein entsprechender SMB-Server für die Windows-Welt leicht nachzurüsten ist.

Als Backup-Geräte sollen sowohl Bandroboter (insbesondere der Sony-Bandroboter auf **LINDE** der Abteilung BS), CD-Laufwerke (evtl. -Roboter) und Einzel-DAT-Laufwerke unterstützt werden. In der Konfigurationsdatei sollten die verschiedenen Medien-Typen (mit Parametern wie z.B. Kapazität) definierbar sein, ebenso die Fähigkeiten der Laufwerke, z.B. welche Medien-Typen darin geschrieben oder gelesen werden können. Zur Steuerung von SCSI-Robotern wird das generische Kommandozeilen-Programm `mtx` verwendet, das generische Roboter mit mehreren eingebauten Laufwerken unterstützt (wobei verschiedene Laufwerke unterschiedliche Fähigkeiten haben können). Der Datentransfer sollte in Abhängigkeit vom eingelegten und möglichst automatisch erkannten Medientyp über ein konfigurierbares externes Programm (z.B. `cp` oder `dd` bzw. `mkisofs/cdrecord`) erfolgen, dabei soll auch die Möglichkeit berücksichtigt werden, daß die Dateien vor der Sicherung von einem weiteren Programm (z.B. zur Kompression oder Verschlüsselung) bearbeitet werden können.

Wünschenswert wäre eine Verwaltungsschnittstelle, die über einen WEB-Browser bedient werden kann. Eine Konfiguration „von Hand“ soll aber dennoch möglich sein.

Als Grundlage für die Arbeit sollten möglichst externe Komponenten verwendet werden, die der BSD- oder GPL-Lizenz (Open Source) unterliegen.

Das Ergebnis der Arbeit soll unter GPL veröffentlicht werden. Der Code sollte vorzugsweise in Perl, Java oder C mit `GNU autoconf` portabel geschrieben werden. Die Firma **daemons point GmbH** und die **Universität Stuttgart** erhalten außerdem das Recht, das Programm unabhängig von der unter GPL veröffentlichten Version zu vermarkten und weiterzuentwickeln

Anhang C

Erklärung

Ich versichere, daß ich diese Arbeit selbstständig verfaßt und nur die angegebenen Hilfsmittel verwendet habe.

Ein Musterexemplar liegt bei der die Arbeit entgegennehmenden Stelle zur Einsicht aus.