Bachelorarbeit

Implementierung eines Dateisystems für den transparenten Zugriff auf ein Versionskontrollsystem

Jens Michael Nödler

13. September 2005

Betreut durch Prof. Dr. Grabowski
Institut für Informatik
Georg-August-Universität Göttingen

Zusammenfassung

Diese Bachelorarbeit beschäftigt sich mit der Implementierung eines Dateisystems für den transparenten Zugriff auf das Versionskontrollsystem Subversion mit Hilfe des WebDAV-Protokolls. Es werden Grundlagen der Versionskontrolle, des WebDAV-Protokolls und der Implementierung von Dateisystemen für das Linux-Betriebssystem vermittelt und das Dateisystem-Framework FUSE vorgestellt. Die Implementierung des Dateisystems und dessen Einsatzmöglichkeiten werden detailliert behandelt. Das implementierte Dateisystem unterstützt grundlegende Dateisystemoperationen ebenso wie den Zugriff auf alle Revisionen eines Subversion-Repositorys und das Sperren von Dateien zum Schutz vor dem Überschreiben von Daten bei Gruppenarbeit.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Versionskontrolle ist heute fester Bestandteil der Softwareentwicklung [34, Chapter 29]. Sie ermöglicht effektive Gruppenarbeit an gemeinsamen Quelltexten, die Pflege mehrerer Versionszweige einer Software, einfaches Nachvollziehen von Änderungen und den Vergleich verschiedener Versionsstände.

Auch außerhalb der Softwareentwicklung sind dies erwünschte Eigenschaften, sobald mehrere Personen an gemeinsamen Daten arbeiten. Dies trifft zum Beispiel bei der Erstellung von Büchern, Studien und Websites zu [31]. Doch häufig wird der vermeintliche Aufwand des Einsatzes eines Versionskontrollsystems gescheut oder die Möglichkeiten der Versionskontrolle sind gänzlich unbekannt.

Ein Versionskontrollsystem als Dateisystem einzubinden, stellt eine Möglichkeit dar, den Einsatz von Versionskontrollsystemen auch außerhalb der Softwareentwicklung zu vereinfachen. Ein solches Dateisystem sollte den transparenten Zugriff auf gemeinsam genutzte Daten ermöglichen. Änderungen sollten für den Anwender transparent durch das Versionskontrollsystem verwaltet und gespeichert werden. Anwender könnten die Vorteile der Versionskontrolle nutzen, ohne ihre gewohnte Arbeitsweise zu ändern und ohne die Bedienung eines Versionskontrollsystems zu erlernen.

An ein Dateisystem für den transparenten Zugriff auf ein Versionskontrollsystem werden folgende Anforderungen gestellt:

- dateisystemgleicher Zugriff auf das Versionskontrollsystem,
- automatische Erzeugung einer neuen Version bei Änderungen,
- effektiver Zugriff auf alle Versionen der Daten,
- Schutz vor gegenseitigem Überschreiben von Daten bei Gruppenarbeit,
- Netzwerkzugriff auf das Versionskontrollsystem.

Diese Bachelorarbeit beschäftigt sich mit der Implementierung eines Dateisystems, das diese Anforderungen erfüllt und stellt dessen Funktionen und Einsatzmöglichkeiten vor.

Als Versionskontrollsystem kommt *Subversion* zum Einsatz und für den Netzwerkzugriff auf Subversion wird das standardisierte *WebDAV*-Protokoll benutzt. Grundlagen der Versionskontrolle und des WebDAV-Protokolls werden im 2. Kapitel behandelt. In diesem Kapitel werden ebenfalls Grundlagen der Implementierung von Dateisystemen für das Linux-Betriebssystem vermittelt und das Dateisystem-Framework *FUSE* vorgestellt. Im 3. Kapitel wird auf die Architektur des Dateisystems für den transparenten Zugriff auf das Versionskontrollsystem Subversion eingegangen. Dort wird ein Überblick über die eingesetzten Technologien gegeben und deren Zusammenspiel erläutert. Das 4. Kapitel beschreibt detailliert die Implementierung des Dateisystems auf Basis von FUSE und besondere versionskontrollspezifische Funktionen des Dateisystems. Im 5. Kapitel werden Einsatzmöglichkeiten eines solchen Dateisystems diskutiert und die wichtigsten Ergebnisse dieser Bachelorarbeit abschließend im 6. Kapitel zusammengefasst.

Kapitel 2

Grundlagen

2.1 Versionskontrolle

Erst der Einsatz von Versionskontrollsoftware ermöglicht effektive Gruppenarbeit mit gemeinsam zu bearbeitenden Daten. Durch zentrale Datenhaltung und Versionierung der Daten werden Änderungsverfolgung, die Pflege mehrerer Versionszweige, das Wiederherstellen und das Vergleichen verschiedener Versionsstände ermöglicht. Der primäre Anwendungsbereich der Versionskontrolle liegt in der Quelltextverwaltung bei der Softwareentwicklung. Es existieren verschiedene technische Ansätze zur Realisierung von Versionskontrollsystemen, deren Gemeinsamkeit in einer zentralen Datenhaltung (engl. *Repository*) liegt. Zur Bearbeitung der Daten muss zuerst eine lokale Kopie (engl. *Working Copy*) des Repositorys erzeugt werden, was als *Checkout* bezeichnet wird [41] [44].

Bei der Änderungsübermittlung in Repositories sind der zentrale und dezentrale Ansatz zu unterscheiden, wobei bei dem zentralen Ansatz die Änderungen durch die Anwender direkt in das Repository zurückgeführt werden. Jeder Anwender hat, entsprechend seiner Zugriffsrechte, die Möglichkeit die Daten des Repositorys einzusehen und Änderungen vorzunehmen. Zu diesen Systemen gehören die bekannten Versionskontrollsysteme *Concurrent Versions System* (CVS) [10] und *Subversion* (SVN) [35].

Das kommerzielle Versionskontrollsystem *Rational ClearCase* von IBM [23] orientiert sich ebenfalls am zentralen Ansatz und bietet als Besonderheit die Möglichkeit eines Dateisystemzugriffs auf die verwalteten Daten. ClearCase-Repositories tragen den Namen *Versioned Object Base* (VOB) und Anwender arbeiten mit Sichtweisen (engl. *Views*) auf diese VOBs. Das *Multiversion File System* (MVFS) von ClearCase erlaubt VOBs als Dateisystem einzubinden und Änderungen an den Daten vorzunehmen. Die Änderungen werden allerdings für jeden Anwender separat gespeichert und nachträglich in das VOB zurückgeführt, so dass jeder Anwender Änderungen vornehmen kann, ohne das zentrale Repository zu verändern [42]. Die Nachteile

von ClearCase liegen in der komplexen Administration, schlechten Performance und den hohen Lizenzkosten.

Der dezentrale Ansatz sieht ebenfalls ein Repository vor, von dem sich Anwender eine lokale Kopie erzeugen können, um daran Änderungen vorzunehmen. Der entscheidende Unterschied zu dem zentralen Ansatz liegt darin, wie Änderungen wieder in das Repository zurückfließen: sie werden als *Changesets* zum Beispiel auf einer Mailingliste veröffentlicht und die Projektleiter entscheiden, welche Änderungen in das Repository übernommen werden. Bekannte Vertreter dieses Ansatzes sind *GNU arch* [17] und *GIT* [15], welches für die Entwicklung des Linux-Kernels eingesetzt wird.

Wenn Daten von mehr als einer Person gleichzeitig bearbeitet werden können, müssen Vorkehrungen getroffen werden, die das gegenseitige Überschreiben von Daten und das Lesen von Daten in einem inkonsistenten Zustand verhindern. Der *Lock-Modify-Unlock*-Prozess sieht vor, dass vor dem Ändern von Daten der Schreibzugriff für Dritte gesperrt wird, so dass nur eine Person diese Daten verändern kann. Die Sperre wird erst nach dem Schreibvorgang entfernt. Der Nachteil bei diesem Vorgehen ist, dass Schreibzugriffe serialisiert stattfinden müssen, obwohl dies nicht immer erforderlich ist, falls die gesperrten Daten nicht geändert wurden.

Eine weitere Strategie ist der *Copy-Modify-Merge*-Prozess, dessen erster Schritt ein Checkout der Daten ist. Nach dem Ändern der Daten wird überprüft, ob die Daten im Repository seit dem Checkout Änderungen durch Dritte erfahren haben. Ist dies der Fall, so müssen die Änderungen aus dem Repository mit den eigenen Änderungen konfliktfrei zusammengeführt werden, um Inkonsistenzen zu vermeiden. Dies ist der so genannte *Merge*-Schritt. Erst nach dem Auflösen aller Konflikte, werden die Änderungen wieder in das Repository übertragen, was als *Checkin* oder *Commit* bezeichnet wird.

Um den Speicherbedarf von Repositories niedrig zu halten, speichern die meisten Systeme nur den jeweiligen Unterschied zur Vorgängerversion an Stelle der kompletten Daten. Der benötigte Speicherplatz hängt vom Umfang der Änderungen und nicht von der Größe der bearbeiteten Daten ab. Bei jedem Zugriff die Daten müssen die Dateien wieder zusammengesetzt werden. Die meisten Algorithmen, die für diesen Zweck eingesetzt werden, sind darauf optimiert, den schnellsten Zugriff auf die jeweils aktuelle Version zu erlauben, so dass der Zugriff auf ältere Versionen länger dauert.

Geläufige Synonyme für Versionskontrolle sind *Versionsverwaltung* bzw. *Version Control System* (VCS) oder *Source Code Managementsystem* (SCM) im englischsprachigen Bereich.

Im Folgenden wird das Versionskontrollsystem Subversion vorgestellt, welches alle Voraussetzungen für die Implementierung eines transparenten Dateisystemzugriffs erfüllt.

2.1.1 Versionskontrolle mit Subversion

Subversion (SVN) ist ein Open-Source-Versionskontrollsystem, das unter den Bedingungen der *Subversion License* [36][1] frei verfügbar ist und als Ersatz für das bekannte Versionskontrollsystem CVS entwickelt wurde, da dieses wichtige Funktionen wie das Löschen und Umbenennen von Dateien nicht unterstützt.

Da sich Subversion an CVS orientiert, basiert es ebenfalls auf dem Copy-Modify-Merge-Prozess und bietet folgenden Funktionsumfang:

- Programme für das Anlegen und Verwalten von Repositories,
- Subversion-Server für den Netzwerkzugriff auf Repositories,
- Apache-Modul für den Netzwerkzugriff per WebDAV,
- kommandozeilenorientierte Applikationen für das Arbeiten mit Repositories (Checkout, Commit, Vergleichen von Versionen, ...)

Subversion bietet neben den von CVS bekannten Funktionen folgende Verbesserungen:

- Versionierung von Verzeichnissen und Metadaten, wie Zugriffsrechten und Zeitstempeln,
- Löschen, Verschieben und Umbenennen von Dateien und Verzeichnissen,
- atomare Commits, was inkonsistente Zustände des Repositorys verhindert,
- Netzwerkzugriff über Apache und WebDAV oder einen eigenen Subversion-Server,
- effiziente Speicherung von Binärdaten.

Subversion versioniert Daten eines Repositorys mit einer globalen Revisionsnummer. Diese Nummer wird bei jeder Änderung des Repositorys inkrementiert, daher erhält jeder Zustand eines Repositorys eine eigene Revisionsnummer. Dies hat den Vorteil, dass jeder Zustand eine eindeutige Nummer besitzt und den Nachteil, dass eine Datei mit zwei verschiedenen Revisionsnummer sich inhaltlich nicht unterscheiden muss, da sich die Revisionsnummer auf Grund einer Änderung einer anderen Datei erhöht hat. Im Gegensatz zu diesem Versionierungsschema benutzt CVS für jede Datei des Repositorys eine eigene Versionsnummer, die nur dann erhöht wird, wenn die entsprechende Datei geändert wurde. Dieses Schema ermöglicht hingegen keinen Zugriff auf alle Daten, die zu einer bestimmten Revisionsnummer gehören, sondern nur den Zugriff auf den Zustand eines Repositorys eines bestimmten Zeitpunkts.

[1] Die *Subversion License* basiert auf der *Apache Software License* [2].

2.1.2 Zugriff auf Subversion-Repositories

Drei Möglichkeiten des Zugriffs auf Subversion-Repositories sind zu unterscheiden:

1. lokaler Zugriff über das `file://`-Schema,
2. Netzwerkzugriff über das `svn://`-Schema mit Hilfe eines eigenen Subversion-Servers (bzw. über das `svn+ssh://`-Schema für einen verschlüsselten SSH-getunnelten Netzwerkzugriff),
3. Netzwerkzugriff per WebDAV über das `http://`-Schema mit Hilfe eines Apache-HTTP-Servers und einem WebDAV-Subversion-Modul für Apache (bzw. über das `https://`-Schema für einen SSL-verschlüsselten Zugriff).

Auf die ersten beiden Zugriffsmöglichkeiten soll im Rahmen dieser Bachelorarbeit nicht näher eingegangen werden. Stattdessen wird der Netzwerkzugriff auf Repositories per WebDAV (WebDAV-Grundlagen finden sich in Abschnitt 2.2 auf Seite 10) im Zusammenspiel mit dem Apache-HTTP-Server vorgestellt, da diese Möglichkeit alle Anforderungen (siehe Abschnitt 3.1 auf Seite 27) für die Implementierung eines Dateisystem für den Zugriff auf Subversion-Repositories erfüllt.

Die Einrichtung des Netzwerkzugriffs per WebDAV wird von Collins-Sussman, Fitzpatrick, Pilato [9] detailliert beschrieben [9, Apache HTTP server], so dass an dieser Stelle nur die wichtigsten Schritte vorgestellt werden sollen:

1. Einrichtung des HTTP-Servers Apache:
   Das WebDAV-Subversion-Modul setzt einen Apache-Server [1] ab Version 2.0.49 (Windows ab 2.0.54) voraus. Die Einrichtung unter Debian GNU/Linux nimmt folgender Befehl vor: `aptitude install apache2 apache2-prefork-dev`.
2. Einrichtung der WebDAV-Module für Apache:
   Die Module `dav_module` und `dav_fs_module` werden bereits mit Apache 2 installiert. Die Module `dav_svn_module` und `authz_svn_module` werden von Subversion bereitgestellt und können unter Debian GNU/Linux per `aptitude install libapache2-svn` installiert werden. Dabei ist darauf zu achten, dass Versionen ab 1.2.0 verwendet werden. Alternativ können die Module auch selbst übersetzt werden. Dazu muss der Subversion-Quelltext[2] geladen und mit folgendem Befehl kompiliert und installiert werden: `./configure --with-ssl --with-apxs=/usr/bin/apxs2 && make && make install`.

[2] `http://subversion.tigris.org/downloads/`

3. Anpassen der Apache-Konfiguration:
   Im letzten Schritt muss die Apache-Konfigurationsdatei `httpd.conf` angepasst werden:

Listing 2.1: Apache Konfigurationsdatei `httpd.conf`

```
1  ServerName localhost
2
3  LoadModule dav_module        /lib/apache2/mod_dav.so
4  LoadModule dav_fs_module     /lib/apache2/mod_dav_fs.so
5  LoadModule dav_svn_module    /lib/apache2/mod_dav_svn.so
6  LoadModule authz_svn_module /lib/apache2/mod_authz_svn.so
7
8  <Location /svn>
9      DAV svn
10     SVNParentPath /path/to/svn-repositories/
11     AuthType Basic
12     AuthName "Subversion Repositories"
13     AuthUserFile /etc/svn-auth-file
14     Require valid-user
15 </Location>
```

   Der Pfad zu den Modulen in den Zeilen 3 bis 6 muss der Systemumgebung angepasst werden und lautet unter Debian GNU/Linux `/usr/lib/apache2/modules/`. Gleiches gilt für den Pfad zu den Subversion-Repositories in Zeile 10. Zu beachten ist, dass der Apache-Prozess häufig unter einem eigenen Benutzerkonto läuft, dem Zugriffsrechte auf das Repository zu gewähren sind.
   Die *Basic HTTP Authentication* (Zeilen 11 bis 14) verhindert anonyme WebDAV-Zugriffe auf die Repositories und bietet einen trivialen Zugriffschutz. Vor dem ersten WebDAV-Zugriff muss ein neuer Benutzer mit dem Kommando `htpasswd -cm /etc/svn-auth-file benutzername` angelegt und ein entsprechendes Passwort vergeben werden. Die Einrichtung einer sicheren SSL-verschlüsselten Verbindung und die differenzierte Vergabe von Zugriffsrechten wird ebenfalls in [9] beschrieben.

Nach einem Neustart des Apache-Servers ist der Subversion-Zugriff mit jedem Webbrowser lesend und einem WebDAV-Client oder den Subversion-Kommandozeilenprogrammen schreibend unter der URI `http://server/svn/repository/` nach der Angabe des Benutzernamens und Passwortes möglich.

Die Betriebssysteme Microsoft Windows XP und Apple Mac OS X sind standardmäßig für den Zugriff auf WebDAV-Server vorbereitet. Unter Windows kann die *Web Folder*-Funktion benutzt werden, während der Mac OS X *Finder* integrierten WebDAV-Zugriff unterstützt. Für Windows existiert zusätzlich eine große Anzahl weiterer WebDAV-Clients wie *Novell NetDrive* oder *SRT WebDrive*. Für Unix- und Linuxbetriebssysteme bieten die Dateimanager der Desktops *Gnome* und *KDE* integrierte Zugriffsmöglichkeiten

auf WebDAV-Server. Das Programm `cadaver` [7] empfiehlt sich für Kommandozeilenanwender.

2.1.3 Neuerungen in Subversion 1.2

Die aktuelle Version 1.2 von Subversion führt einige Neuerungen ein, die besonders die Implementierung eines Dateisystems für den Zugriff auf Subversion-Repositories erleichtern.

Die erste Neuerung ist die so genannte Autoversionierung, die bei jeder Änderung am Repository zur Erzeugung einer neuen Revision führt. Diese Funktion ist nur für den Netzwerkzugriff per WebDAV verfügbar, da es sich um eine Funktionalität der WebDAV-Erweiterung *DeltaV* gemäß RFC 3253 [8] (engl. *Request for Comments*) handelt. Details zu DeltaV finden sich in Abschnitt 2.2.4 auf Seite 15 dieser Arbeit. Um die Autoversionierung zu aktivieren, muss der Eintrag `SVNAutoversioning on` in der Apache-Konfigurationsdatei `httpd.conf` innerhalb der `<Location>`-Direktive hinzugefügt und der Apache-Server neu gestartet werden. Fortan führt jede von einem WebDAV-Client durchgeführte Änderung automatisch zu einer neuen Revision. Diese Option kann für die Implementierung eines Dateisystems für den transparenten Zugriff auf Subversion genutzt werden, da kein gesonderter Checkout/Commit-Zyklus erforderlich ist, um eine neue Revision zu erzeugen.

Zu beachten ist, dass die Autoversionierung das Verhalten bei Checkout/Commit-Zyklen mit den Subversion-Programmen für das Arbeiten mit Repositories nicht verändert. Sie kommt nur bei dem Zugriff mittels WebDAV-Clients zum tragen, die Änderungen an den Daten vornehmen. So ist eine konfliktfreie Koexistenz beider Zugriffsvarianten auf Subversion-Repositories möglich.

Die zweite wichtige Neuerung von Subversion 1.2, ist die Unterstützung für das exklusive Sperren von Dateien (engl. *Locking*). Somit unterstützt Subversion nicht nur den Copy-Modify-Merge- sondern auch den Lock-Modify-Unlock-Prozess. Das Sperren von Dateien ist in Kombination mit der Autoversionierung sinnvoll, da Daten vor dem Ändern gesperrt werden können. Dies verhindert Inkonsistenzen und gegenseitiges Überschreiben von Daten bei Gruppenarbeit. Das Sperren von Dateien ist ebenfalls über WebDAV-Verbindungen verfügbar, da Locking Bestandteil der WebDAV-Spezifikation ist. Auch diese Funktion kann bei der Implementierung eines Dateisystem für den Subversion-Zugriff genutzt werden, um Dateien während der Bearbeitung sperren zu können.

2.2 WebDAV-Protokoll

Das Akronym *WebDAV* steht für *Distributed Authoring and Versioning Protocol for the World Wide Web* und ermöglicht das Lesen und Schreiben von entfernten Daten. Der erste Schritt der Standardisierung des Protokolls bestand im Festlegen der Anforderungen, welche von einer IETF-Arbeitsgruppe im RFC 2291 [33] (engl. *Request for Comments*) beschrieben wurden. Die Arbeitsgruppe entschloss sich auf Grund des Umfangs des Vorhabens, die WebDAV-Spezifikation in zwei Teile zu gliedern und nacheinander zu veröffentlichen. Zuerst wurden 1999 im RFC 2518 [21] die Aspekte des verteilten Arbeitens (HTTP Extensions for Distributed Authoring) veröffentlicht und im Jahr 2002 folgte der Aspekt der Versionierung im RFC 3253 [8] (Versioning Extensions to WebDAV).

Da WebDAV auf dem *Hypertext Transfer Protocol* (HTTP) aufbaut, werden im Folgenden grundlegende HTTP-Funktionsweisen vorgestellt, um anschließend auf das WebDAV-Protokoll einzugehen.

2.2.1 HTTP-Grundlagen

Das Hypertext Transfer Protokoll ist das Standardprotokoll zur Kommunikation zwischen Webserver und Webbrowser im Internet. Bei HTTP handelt es sich um ein universell verwendbares Protokoll der Anwendungschicht des TCP/IP-Protokollstacks, dessen aktuelle Version HTTP/1.1 im RFC 2616 [11] standardisiert ist. Grundlegend ist, dass das Protokoll in einem Anfrage-/Antwortzyklus arbeitet, wobei ein Client eine Anfrage sendet, die von einem HTTP-Server beantwortet wird.
Eine Anfrage enthält immer eine HTTP-Methode (Zeile 1, im Beispiel `HEAD`) und einen *Uniform Resource Identifier* (URI gemäß RFC 3986 [5]):

Listing 2.2: Beispiel: HTTP-Anfrage

```
1 HEAD /datei.html HTTP/1.1
2 Host: www.noedler.de
```

Der Server antwortet mit einem Statuscode (Zeile 1, im Beispiel `302 Found`), Metadaten (Zeilen 2 bis 4) und mit den angeforderten Daten, falls die Anfrage erfolgreich war:

Listing 2.3: Beispiel: HTTP-Antwort

```
1 HTTP/1.1 302 Found
2 Server: Apache/2.0.54 (Debian GNU/Linux)
3 Content-Type: text/html
4 Connection: close
```

HTTP transportiert Status- und Metadaten im Kopf der Nachrichten (engl. *Message Header*) und die Daten im Körper der Nachrichten (engl. *Message Body*). RFC 2616 definiert zulässige Methoden, Statuscodes und Metadaten (engl. *Header Field Definitions*).

Die wichtigsten HTTP-Methoden[3] sind:

- `GET` – Sendet die angeforderten Daten von einem Server zu dem Client,
- `HEAD` – Sendet nur den Nachrichtenkopf der angeforderten Daten,
- `POST` – Erlaubt das Senden von Daten des Clients an den Server,
- `PUT` – Erlaubt das Speichern von Daten des Clients unter einer URI,
- `DELETE` – Erlaubt das Löschen von Daten auf dem Server.

Die wichtigsten Statuscodes[4] sind in fünf Gruppen eingeteilt:

- `Informational 1xx` – Rein informationelle Nachrichten. Zum Beispiel `100 Continue` als Nachricht an den Client, dass die Anfrage vom Server bearbeitet wird.
- `Successful 2xx` – Die Anfrage wurde erfolgreich bearbeitet. Zum Beispiel `200 OK` gefolgt von weiteren Metadaten und den zuvor per `GET` angeforderten Daten.
- `Redirection 3xx` – Weiterleitung einer Anfrage. Zum Beispiel `301 Moved Permanently`, falls eine Ressource eine neue permanente URI besitzt. Die neue URI wird in den Metadaten der Antwort dem Client zugesendet.
- `Client Error 4xx` – Fehler auf Seite des Clients. Zum Beispiel `404 Not Found`, falls eine angeforderte Ressource nicht existiert.
- `Server Error 5xx` – Fehler auf Seite des Servers. Zum Beispiel `501 Not Implemented`, falls die angeforderte Methode vom Server nicht unterstützt wird.

2.2.2 Anforderungen an WebDAV

Das Ziel der IETF-Arbeitsgruppe war die Umsetzung der Idee des Web-Erfinders Tim Berners-Lee, das Web nicht ausschließlich als Lesemedium zu nutzen, sondern einen ebenso einfachen Weg für das Editieren des Webs anzubieten.

Das Internet-Basisprotokoll HTTP sieht bereits seit der Version 1.0 Methoden für das Heraufladen und Löschen (`PUT` und `DELETE`) von Dateien vor [4, Appendix D]. Diese Methoden werden allerdings nur von wenigen HTTP-Servern akzeptiert, so dass sie kaum Anwendung finden.

Diese Schwachstelle von HTTP soll WebDAV ausgleichen. Die Anforderungen an das WebDAV-Protokoll sind daher:

[3] alle HTTP-Methoden: RFC 2616 [11, Section 9, Method Definitions]
[4] alle Statuscodes: RFC 2616 [11, Section 10, Status Code Definitions]

- Laden, Editieren und Löschen von Ressourcen,
- Ressourcen neue Eigenschaften (engl. *Properties*) hinzufügen und vorhandene Eigenschaften ändern und abfragen,
- Zusammengehörigkeiten in Kollektionen (engl. *Collections*) festlegen und abfragen,
- Kollisionsvermeidung bei Gruppenarbeit durch Sperren (engl. *Locking*) von Ressourcen.

Unter Ressourcen sind alle über einen URI identifizierbaren Ziele zu verstehen. Der Begriff der Ressource ist in dem RFC 3986 [5, Section 1] abstrakt definiert: „This specification does not limit the scope of what might be a resource“. Im Kontext von WebDAV kann es sich daher um Dateien oder Verzeichnisse (Kollektionen) handeln.

2.2.3 WebDAV Distributed Authoring

WebDAV soll das Web bidirektional nutzbar machen und dabei gleichzeitig zu dem verbreiteten HTTP kompatibel bleiben. Diese Anforderungen können erfüllt werden, indem WebDAV zusätzliche HTTP-Methoden einführt und einige HTTP-Methoden redefiniert.
RFC 2518 [21] definiert folgende neue HTTP-Methoden:

- `PROPFIND` – Steht für *Property Finding* und ermöglicht das Abfragen von Eigenschaften (Metadaten) einer Ressource.
- `PROPPATCH` – Steht für *Property Patch* und ermöglicht das Hinzufügen, Ändern und Löschen von Eigenschaften einer Ressource.
- `MKCOL` – Steht für *Make Collection* und ermöglicht das Anlegen von Kollektionen mehrerer Ressourcen, was dem Erstellen von Verzeichnissen bei Dateisystemen gleicht.
- `COPY` – Ermöglicht das Kopieren von Ressourcen.
- `MOVE` – Ermöglicht das Verschieben von Ressourcen.
- `LOCK` – Ermöglicht das Sperren von Ressourcen, um Schreibzugriffe Dritter zu verhindern.
- `UNLOCK` – Entfernt eine zuvor gesetzte Sperre.

RFC 2518 [21] redefiniert folgende HTTP-Methoden:

- `PUT` – Sollen Dateien heraufgeladen werden, deren Elternverzeichnisse noch nicht existieren, schlägt dies mit dem Fehlercode `409 Conflict` fehl. Elternverzeichnisse müssen zuvor mittels `MKCOL` erstellt werden.

- `DELETE` – Bei dem Löschen von Dateien müssen die internen Verweise (Eigenschaften des Elternverzeichnisses) auf diese Datei entfernt werden. Sollte die zu löschende Ressource gesperrt sein, wird der Fehler `423 Locked` gesendet.

Die Definitionen der HTTP-Methoden `GET`, `HEAD` und `POST` blieben unverändert, was die Kompatibilität zu HTTP-Clients sicherstellt. WebDAV führt ebenfalls eine Reihe neuer HTTP-Statuscodes ein. Die wichtigsten davon sind:

- `207 Multi-Status` – Die Antwort eines WebDAV-Servers enthält im Nachrichtenkörper XML-Daten (engl. *Extensible Markup Language*), die weitere Statuscodes und Metadaten enthalten.
- `422 Unprocessable Entity` – Der Server kann die Anfrage nicht bearbeiten, da die XML-Daten semantisch fehlerhaft sind.
- `423 Locked` – Eine Ressource ist gesperrt und die angefragte Methode kann zurzeit nicht ausgeführt werden.
- `424 Failed Dependency` – Die angefragte Methode konnte nicht ausgeführt werden, da eine vorhergehende Methode nicht erfolgreich ausgeführt werden konnte.

Von Bedeutung sind folgende neue HTTP-Header:

- `DAV` – WebDAV-Ressourcen sind in zwei Klassen eingeteilt: *Class 1*-Ressourcen müssen HTTP/1.1 kompatibel sein und alle gemäß RFC 2518 [21] als verpflichtend gekennzeichneten Anforderungen erfüllen. *Class 2*-Ressourcen bietet zusätzlich die Möglichkeit Ressourcen zu sperren (engl. *Locking*). Dieser HTTP-Header gibt an welcher Klasse eine Ressource angehört.
- `Depth` – Gibt an, ob Methoden auch auf Mitglieder einer Ressource anzuwenden sind. So sperrt eine `LOCK`-Methode mit dem Depth-Wert `infinity`, die auf eine Kollektion angewandt wird, ebenfalls alle Mitglieder der Kollektion.
- `Lock-Token` – Bei dem Sperren wird dem Client ein eindeutiger Lock-Token mitgeteilt, der bei dem Entsperren mit diesem HTTP-Header dem Server wieder mitgeteilt werden muss.
- `Overwrite` – Legt fest, ob bei den Methoden `COPY` oder `MOVE` die Zielressource überschrieben werden darf.

Im Gegensatz zu HTTP, das Metadaten unstrukturiert im Kopf der Nachricht transportiert, benutzt das WebDAV-Protokoll dazu XML innerhalb des Nachrichtenkörpers, falls die Benutzung einfacher Headern nicht

ausreicht, um die Information zu transportieren. Die Verwendung von XML hat folgende Vorteile: Erweiterbarkeit um neue Eigenschaften mit Hilfe von XML-Namensräumen, beliebiger Umfang von Metadaten und die einfache Weiterverarbeitbarkeit der strukturierten XML-Daten mit entsprechenden Parsern.

An dem Beispiel einer WebDAV-`PROPFIND`-Anfrage soll dieses veranschaulicht werden:

Listing 2.4: Beispiel: Anfrage von zwei WebDAV-Eigenschaften einer Ressource

```
1  PROPFIND /datei HTTP/1.1
2  Host: www.noedler.de
3  Content-type: text/xml; charset="utf-8"
4  Content-Length: xx
5
6  <?xml version="1.0" encoding="utf-8" ?>
7  <D:propfind xmlns:D="DAV:">
8      <D:prop xmlns:N="http://www.noedler.de/schema/">
9          <N:author/>
10         <D:getlastmodified/>
11     </D:prop>
12 </D:propfind>
```

Analog zu HTTP wird im Kopf die Methode und das Ziel der Anfrage festgelegt (Zeilen 1 und 2) sowie der Typ und die Länge des Inhalts des Nachrichtenkörpers mitgeteilt (Zeilen 3 und 4). Innerhalb des Nachrichtenkörpers wird festgelegt, welche Eigenschaften dieser Ressource abgefragt werden sollen und aus welchem XML-Namensraum die Elemente stammen, wobei der Namensraum `DAV:` gemäß RFC 2518 [21] der WebDAV-Standardnamensraum und der Zweite (Zeilen 8 und 9) ein selbstdefinierter Namensraum ist.

Listing 2.5: Beispiel: Antwort der Anfrage der WebDAV-Eigenschaften

```
1  HTTP/1.1 207 Multi-Status
2  Content-Type: text/xml; charset="utf-8"
3  Content-Length: xx
4
5  <?xml version="1.0" encoding="utf-8" ?>
6  <D:multistatus xmlns:D="DAV:">
7      <D:response>
8      <D:href>http://www.noedler.de/datei</D:href>
9      <D:propstat>
10         <D:prop xmlns:N="http://www.noedler.de/schema/">
11             <N:author>Jens M. Noedler</R:author>
12             <D:getlastmodified>
13                 Mon, 08 Aug 2005 15:42:29 GMT
14             </D:getlastmodified>
15             </D:prop>
16         <D:status>HTTP/1.1 200 OK</D:status>
17     </D:propstat>
18     </D:response>
19 </D:multistatus>
```

Die Antwort erfolgt mit dem von WebDAV eingeführten HTTP-Statuscode `207 Multi-Status` und liefert die angefragten Eigenschaften der Ressource im Nachrichtenkörper als XML-Daten.

Auch wenn WebDAV sowohl auf Server- als auch auf Clientseite viel Unterstützung erfahren hat, konnte es sich noch nicht auf breiter Basis durchsetzen, denn auch ohne WebDAV ist es mit der HTTP-Methode `POST` in Verbindung mit HTML-Formularen möglich, entfernte Ressourcen zu editieren (zum Beispiel Wikis und Blogs) und Dateien auf Server zu laden. Jedoch bietet WebDAV Funktionen, die mit HTTP nicht zu erreichen sind. Dazu gehört beispielsweise das gleichzeitige Heraufladen mehrerer Dateien, die Verwaltung von Metadaten (`PROPFIND, PROPPATCH`) und die Möglichkeit der Versionierung von Ressourcen, die im Folgenden vorgestellt wird.

2.2.4 WebDAV Versioning Extensions

Mit Veröffentlichung des RFC 3253 [8] wurde die WebDAV-Spezifikation um Versionskontrolleigenschaften erweitert und somit komplettiert, da RFC 2518 [21] nur das verteilte Bearbeiten von Ressourcen abdeckt. Die Spezifikation, welche auch als *DeltaV* bezeichnet wird, definiert neue Methoden, Header und Eigenschaften, um Versionskontrolle für entfernte Ressourcen nutzbar zu machen.

Im Gegensatz zu den in Abschnitt 2.1 vorgestellten Versionskontrollsystemen arbeitet DeltaV nicht repository- sondern ressourcenorientiert, so dass jede Ressource einzeln versioniert wird und nicht alle Dateien eines Repositorys. So kann zum Beispiel die Ressource `http://server/datei01` versionsverwaltet sein, während es die Ressource `http://server/datei02` nicht sein muss. Obwohl auch Versionskontrollsysteme die Möglichkeiten bieten, bestimmte Dateien oder Dateitypen von der Versionierung auszuschließen, besteht der Unterschied darin, dass der DeltaV-Ansatz kein Repository vorsieht, sondern die Versionierung für jede Ressource einzeln stattfindet.
Daraus ergeben sich folgende Anforderungen an DeltaV:

1. einzelne Ressource unter Versionskontrolle stellen,
2. Herausfinden, ob eine Ressource unter Versionskontrolle steht,
3. Erzeugen von neuer Versionen und Zugreifen auf verschiedene Versionen einer Ressource,
4. Erzeugen von Versionszweigen (engl. *Branch*) und Zusammenführen von Versionszweigen (engl. *Merge*) einer Ressource.

Da per WebDAV Dateien im Netzwerk bearbeitbar sind, besteht keine Notwendigkeit die Daten zuvor von dem Server zu laden. Daher sieht die DeltaV-Spezifikation folgende Möglichkeiten für das Bearbeiten von Ressourcen vor:

1. *Basic-Server-Workspace* – Jeder Client besitzt auf dem Server einen eigenen Arbeitsbereich (engl. *Workspace*) für das Bearbeiten von Ressourcen, nachdem der Client ein Checkout der Ressource in seinen Arbeitsbereich durchgeführt hat. Die Änderungen werden vom Client per Checkin wieder zurückgeschrieben.

2. *Basic-Client-Workspace* – Beschreibt einen ähnlichen Prozess mit dem Unterschied, dass der Arbeitsbereich nicht auf dem Server, sondern auf dem Client gespeichert wird. Beide Prozesse unterstützen den kompletten DeltaV-Funktionsumfang, wie das Anlegen mehrerer Versionszweige und das Zusammenführen von Versionszweigen.

3. *Core-Versioning* – Ist die Minimalversion, die ausschließlich lineare Versionierung ohne Versionszweige erlaubt. Diese Funktion kann auch für WebDAV-Clients ohne DeltaV-Erweiterung verfügbar gemacht werden, so dass die Versionierung einer Ressource nicht durch DeltaV-Anweisungen ausgelöst wird, sondern jede Änderung einer Ressource automatisch zu einer neuen Version führt [8, Section 3.2.2, DAV:auto-version].
   Diese Funktion nutzt auch das WebDAV-Subversion-Modul, wenn die in Abschnitt 2.1.3 beschriebene Autoversionierung aktiviert ist. Dafür wird die DeltaV-Eigenschaft `DAV:auto-version` auf den Wert `DAV:checkout-checkin` gesetzt[5], so dass jeder eine Ressource modifizierende Zugriff automatisch von einem Checkout/Commit-Zyklus eingerahmt wird und eine neue Version erzeugt wird. Dabei muss beachtet werden, dass *jede* Änderung zu einer neuen Revision führt, so dass deutlich mehr Revisionen erzeugt werden, als bei manuellen Checkout/Commit-Zyklen.

Auf Grund der Tatsache, dass für die Implementierung des Dateisystems für den transparenten Subversion-Zugriff die DeltaV-Autoversionierung genutzt wird und die restliche DeltaV-Spezifikation sehr umfangreich und für diese Bachelorarbeit nicht elementar ist, wird auf eine detailliertere Darstellung verzichtet.

[5] Siehe Subversion 1.2.1 Quelltext, Datei `./subversion/mod_dav_svn/liveprops.c`, Zeilen 398 ff.

2.3 Dateisysteme unter Linux

Da es sich bei dem Kernel des Linux-Betriebssystems ebenso wie bei nahezu allen anderen Unix-Systemen um einen monolithischen Kernel handelt [27, Chapter 1] [6, Preface], sind auch Linux-Dateisysteme Teil des Kernels. Linux besitzt eine große Auswahl von Dateisystemen, die entweder als fester Teil des Kernel-Images kompiliert werden oder dynamisch als Kernel-Modul zur Laufzeit nachgeladen werden können.

Die verfügbaren Dateisysteme lassen sich in drei Klassen einordnen: lokale Dateisysteme für den Zugriff auf Festplatten und andere Medien (wie *ext3* oder *UDF*), Netzwerkdateisysteme für das Einbinden entfernter Ressourcen in den lokalen Verzeichnisbaum (wie *Network Filesystem* (NFS) oder *Server Message Block* (SMB)) und virtuelle Dateisysteme (wie */proc* oder *tmpfs*). Wobei letztere keine physikalisch auf einem Datenträger vorhandenen Ressourcen repräsentieren, sondern Daten, die im Hauptspeicher zwischengespeichert sind und im Fall des */proc*-Dateisystems zum Beispiel Informationen über aktive Prozesse oder den Status des Kernels liefern.

Die Aufgabe der lokalen Dateisysteme ist die Verwaltung physikalischer Datenträger [37, Kapitel 10.6.3] und die Abstraktion davon, so dass die Datenträger in Form von Dateien, Verweisen und Verzeichnissen genutzt werden können. Netzwerkdateisysteme setzen auf lokalen Dateisystemen auf, nutzen deren Funktionen zur Verwaltung von Daten und ermöglichen den Zugriff darauf über Netzwerke. Ihre Aufgabe besteht darin, entfernte Ressourcen lokal verfügbar zu machen, also angeforderte Daten der entfernten Ressourcen über das Netzwerk zu transportieren und Änderungen der Daten wieder zuverlässig zurückzutransportieren. Dies soll für den Anwender eines Netzwerkdateisystems möglichst transparent stattfinden, so dass kein Unterschied zur Nutzung eines lokalen Dateisystems besteht.

Allen Dateisystemen des Linux-Kernels ist gemeinsam, dass sie innerhalb des Kernels implementiert sind und als Teil des Kernels – im *Kernelspace* – ausgeführt werden. Für Quelltexte des Kernels gelten unter anderem folgende Bedingungen [27, Chapter 2]:

- Benutzung von *GNU C* anstatt des standardisierten *ANSI C* als Programmiersprache. GNU C wird von der *GNU Compiler Collection* (GCC) [14] implementiert und stellt eine Erweiterung des ANSI C- bzw. ISO C-Standards dar [40].

- Kein Zugriff auf Standard-C-Bibliotheken (wie *libc* [18]) oder andere Bibliotheken, da diese Teil des Benutzermodus (engl. *Userspace*) sind. Der Kernel stellt nur einen kleinen Teil der *libc*-Funktionen wie zum Beispiel solche zur String-Verarbeitung zur Verfügung, in dem die Methoden innerhalb des Kernels selbst implementiert wurden.

- Während Userspace-Programme bei einem ungültigen Speicherzugriff mit dem `SIGSEGV`-Signal vom Kernel beendet werden können, kennt

der Kernel selbst keine Speicherschutzmechanismen. So würde ein fehlerhafter Speicherzugriff innerhalb des Kernel zu einem Systemabsturz (*Kernel Oops*) führen.

- Der Linux-Kernel ist präemptiv, so dass ausgeführter Code durch anderen unterbrochen werden kann, zum Beispiel weil dieser eine höhere Priorität besitzt. Die Ausführung von Code kann zusätzlich durch auftretende Interrupts unterbrochen werden, die bevorzugt bearbeitet werden müssen. Daher muss sichergestellt werden, dass es zu keiner *Deadlock*-Situation oder Inkonsistenz kommen kann.

- Durch die Multiprozessor- und Multitaskingfähigkeiten des Kernels muss Kernel-Code eigene Ressourcen vor dem Zugriff Dritter schützen, um die Konsistenz der Daten zu gewährleisten.

Diese Besonderheiten machen die Entwicklung von Kernel-Code im Gegensatz zu Programmen für den Benutzermodus sehr aufwändig. Zusätzlich bedarf es umfangreicher Test, da der Kernel-Code auch auf verschiedenen Plattformen lauffähig sein muss und ein Fehler größere Auswirkung hätte als im Benutzermodus.

Eine weitere Eigenschafte der Benutzung von Dateisystemen des Linux-Kernels ist, dass diese grundsätzlich nur von dem Systemverwalter *root* oder einem privilegierten Prozess in den lokalen Verzeichnisbaum eingebunden werden können. Ausnahmen hiervon bilden das Konzept des *Automounts*, wobei Dateisysteme automatisch eingebunden werden, sobald diese von einem Anwender benötigt werden oder ein entsprechender Eintrag in der Konfigurationsdatei `/etc/fstab`, der Anwendern das manuelle Einbinden von Dateisystemen erlaubt. Da diese Optionen nicht immer standardmäßig eingerichtet sind, ist immer der Eingriff des Systemverwalters notwendig, um Anwendern das Einbinden von Dateisystemen zu ermöglichen.

2.3.1 Konzept des Virtual Filesystem

Um einen einheitlichen Zugriff auf jede Art von Dateisystem zu ermöglichen, benutzt Linux das Konzept des *Virtual Filesystem* (VFS), welches die Standardschnittstelle für den Zugriff von Programmen auf Dateisysteme darstellt [6, Chapter 12] [27, Chapter 12]. Jede Dateisystemoperation erscheint für Programme durch die zusätzliche Abstraktionsschicht gleichartig, egal auf welches konkrete Dateisystem zugegriffen wird. Das VFS leitet die Aufrufe an das jeweils zuständige Dateisystem weiter.

Kopiert ein Anwender beispielsweise eine Datei von einem Netzwerklaufwerk, welches per NFS eingebunden ist, auf eine lokale Festplatte, welche von dem Dateisystem ext3 verwaltet wird, so bemerkt er diesen Unterschied nicht. Erst diese Abstraktion und die daraus resultierende Transparenz bei der Benutzung ermöglicht einen definierten Befehlssatz von Dateisystemope-

rationen auf der einen und eine Vielzahl verschiedener Dateisysteme auf der anderen Seite.

Detaillierter betrachtet verläuft der Kopiervorgang im Beispiel einer Datei von einer NFS-Quelle auf ein ext3-Laufwerk so, dass das kopierende Programm die Quelldatei lesend und die Zieldatei schreibend öffnet. Diese generischen Aufrufe werden vom VFS an die entsprechenden Methoden des jeweiligen Dateisystems delegiert, so dass für einen solchen Kopiervorgang die `read()`-Methode des NFS-Dateisystems und die `write()`-Methode des ext3-Dateisystems aufgerufen wird.
Folgende Grafik zeigt den schematischen Aufbau des Virtual Filesystems:

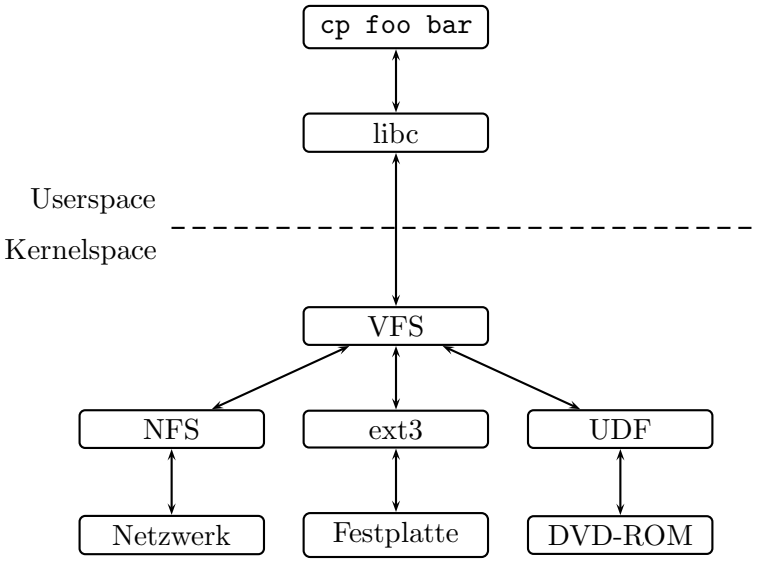

Abbildung 2.1: Konzept des Virtual Filesystem im Linux-Kernel

Diese Grafik verzichtet die auf Darstellung der Kernelschicht der Systemaufrufe (engl. *System Calls*, kurz *Syscalls*), welche die Schnittstelle zwischen Benutzermodus und Kernel darstellt [27, Chapter 5] [6, Chapter 8]. Die Zugriffe aus dem Benutzermodus auf den Kernel sind grundsätzlich nur mit Systemaufrufen möglich, wobei die Benutzung von Syscalls durch die Verwendung der Standard-C-Bibliotheken (wie *libc*) erleichtert wird. Um die Übersichtlichkeit zu wahren und da nur ein kleiner Teil der Systemaufrufe dateisystemspezifisch sind[6], wird auf eine genauere Darstellung verzichtet.

Das Virtual Filesystem bietet nicht nur für Applikationen eine einheitliche Schnittstelle, sondern auch die Dateisysteme des VFS müssen sich an Standards halten. Das VFS definiert daher Datenstrukturen, die ein Dateisystem implementieren muss, um mit dem VFS kompatibel zu sein. Die vier wichtigsten Datenstrukturen, die auch VFS-Objekte genannt werden, sind:

[6]Eine Liste der Systemaufrufe des Linux-Kernels für die i386-Architektur findet sich im Linux-Quelltext in der Datei `./arch/i386/kernel/syscall_table.S`.

1. *Superblock*, welcher ein eingebundenes Dateisystem repräsentiert.
2. *Inode* (Index Node), welcher eine Datei des Dateisystems repräsentiert.
3. *Dentry* (Directory Entry, Verzeichniseintrag), welcher einen Teil eines Pfades repräsentiert. Beispiel: Der Pfad `/home/user/datei` besteht aus den Verzeichniseinträgen (Dentries) `/`, `home`, `user` und `datei`.
4. *File*, welcher eine von einem Prozess geöffnete Datei repräsentiert.

Jede dieser Datenstrukturen ist in C als `struct` definiert und besitzt neben Datenfeldern, die Eigenschaften des Objekts definieren, weitere `struct`-Datenstrukturen. In einer dieser `struct`-Datenstrukturen sind Operationen definiert, die auf dem entsprechenden VFS-Objekt ausgeführt werden können. So besitzt das Superblock-Objekt eine `super_operations`-Datenstruktur, welche Operationen wie `read_inode()` oder `remount_fs()` definiert. Da es sich bei diesen Operationen nur um die Schnittstelle und nicht um die Implementierung handelt, sind sie als *Callback*-Methoden [22] definiert, so dass jedes Dateisystem seine spezifische Implementierung vornehmen bzw. nicht implementierte Methoden auf `NULL` setzen kann.

Die Inode-Datenstruktur `struct inode` definiert Eigenschaften von Dateien und Verzeichnissen, da diese – der Unix-Idee „Alles ist eine Datei“ entsprechend – gleich behandelt werden. Zu diesen Eigenschaften zählen zum Beispiel Eigentümer, Zugriffsrechte und Erstellungs- und Änderungszeitpunkt. Die Operationen eines Inode-Objekts sind in `struct inode_operations` definiert und sind die von Unix/Linux bekannte Methoden zum Bearbeiten von Dateien und Verzeichnissen wie `unlink()`, `rename()` oder `rmdir()`. Allen hier definierten Methoden ist gleich, dass sie auf nicht geöffneten Dateien operieren. Für von einem Prozess geöffnete Dateien ist das `struct file`-Objekts vorgesehen, welches das `struct file_operations`-Objekt definiert, das Methoden wie `open()`, `read()` und `release()` bereitstellt.

Hier ein Auszug der `struct file_operations`-Datenstruktur mit einigen wichtigen Operationen:

Listing 2.6: Auszug der `struct file_operations`-Datenstruktur

```
1  struct file_operations {
2      int (*open)
3          (struct inode *, struct file *);
4      ssize_t (*read)
5          (struct file *, char __user *, size_t, loff_t *);
6      ssize_t (*write)
7          (struct file *, const char __user *, size_t, loff_t *);
8      int (*readdir)
9          (struct file *, void *, filldir_t);
10 };
```

Die vollständigen `struct`-Datenstrukturen finden sich im Linux-Kernel-Quelltext in den Dateien `./include/linux/fs.h` und `./include/linux/dcache.h`.

2.3.2 Filesystem in Userspace (FUSE)

Das *Filesystem in Userspace* (FUSE) [12] ist eine Erweiterung für das Linux-Betriebssystem, die es ermöglicht, Dateisysteme für den Benutzermodus (engl. *Userspace*) zu implementieren, sie im Benutzermodus auszuführen und das Einbinden solcher Dateisysteme durch nicht privilegierte Benutzer zu erlauben. FUSE selbst ist kein Dateisystem, sondern ein Framework zu Implementierung und Nutzung von Dateisystemen, die im Benutzermodus ausgeführt werden. FUSE ist für die Linux-Kernel der Serien 2.4 und 2.6 unter den Bedingungen der *GNU General Public License* (GPL) [19] verfügbar.
Das FUSE-Framework besteht aus drei Komponenten:

1. Linux-Kernel-Modul (`fuse.ko`), welches für die Interaktion mit der Dateisystemschicht des Kernels sorgt und sich unterhalb der VFS-Schicht einklinkt und die Zugriffe auf FUSE-basierte Dateisysteme an die FUSE-Bibliothek weiterreicht.

2. FUSE-Bibliothek (`libfuse`) für den Benutzermodus, die grundlegende Funktionen für FUSE-basierte Dateisysteme zur Verfügung stellt. Die FUSE-Bibliothek steht unter der für Bibliotheken gedachten *GNU Lesser General Public License* (LGPL) [20].

3. Benutzermodusprogramm (`fusermount`) für das Ein- und Ausbinden (engl. *mount/unmount*) von FUSE-basierten Dateisystemen. Der Name des Programmes setzt sich aus den Wörtern *fuse user mount* zusammen.

Die Installation von FUSE beschränkt sich nach dem Entpacken auf den Befehl `./configure && make && make install`. Das FUSE-Kernel-Modul

kann auch als fester Teil des Kernel-Images kompiliert werden, wenn an Stelle des zurzeit stabilen Kernels v2.6.13 der von Andrew Morton gepflegte Linux-Entwickler-Kernel (mm-Serie, zurzeit v2.6.13-mm3) genutzt wird. Da FUSE ab v2.6.14 auch in der Serie der stabilen Kernel verfügbar sein wird[7], kann auch die aktuelle Entwicklerversion v2.6.14-rc1 benutzt werden.

Folgende Grafik verdeutlicht die FUSE-Architektur und die Aufgabe des FUSE-Moduls, welches innerhalb des Kernels Aufrufe für FUSE-Dateisysteme an die FUSE-Bibliothek im Benutzermodus weiterreicht.

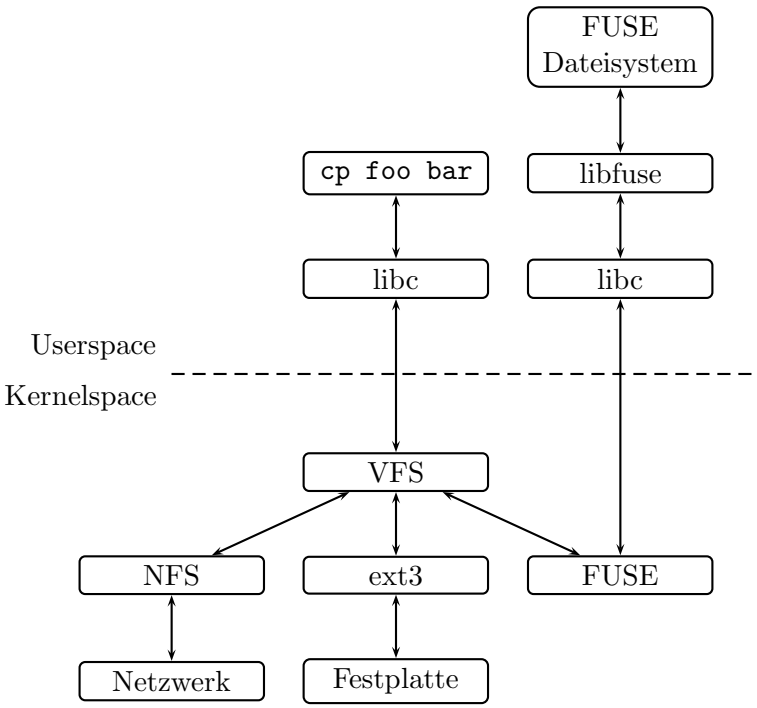

Abbildung 2.2: FUSE-Architektur: VFS und Userspace im Zusammenspiel

Die Benutzung eines FUSE-Dateisystems erfolgt dabei aus Sicht eines Anwenders so, dass zuerst das Dateisystem eingebunden wird. Zum Beispiel lautet der Aufruf für das Einbinden des FUSE-basierten Dateisystems *sshfs* [13], welches per SSH entfernte Systeme lokal einbinden kann: `sshfs benutzer@rechner:/einzubindender/pfad/ /tmp/mountpoint`. Nach der erfolgreichen SSH-Anmeldung am entfernten System, ist der eingebundene Pfad unter `/tmp/mountpoint` zugreifbar und der Anwender kann zum Beispiel Dateien auf dem entfernten System ändern oder mit dem lokalen System austauschen. Der Vorteil besteht darin, dass er keine zusätzlichen Programme für das Kopieren von Daten per SSH benutzen muss, da dies für den

[7] kernel status, 5 Sep 2005
`http://www.ussg.iu.edu/hypermail/linux/kernel/0509.0/1387.html`

Anwender transparent im Hintergrund geschieht. Mit Hilfe des Aufrufs `fusermount -u /tmp/mointpoint` kann das Dateisystem wieder entfernt werden.

Aus Sicht der Dateisystemaufrufe werden diese wie gewohnt von dem Benutzerprogramm über die Standardbibliothek *libc* an den Kernel weitergereicht. Dort reicht das VFS die Aufrufe an das zuständige Dateisystem weiter. Das in diesem Fall zuständige FUSE-System leitet die Aufrufe zur FUSE-Bibliothek wieder zurück in den Benutzermodus. Die Bibliothek reicht diese wiederum an das FUSE-basierte Dateisystem weiter. In dem Fall von *sshfs* würde der ursprüngliche Aufruf des Anwenders hier in SSH-Befehle umgesetzt, die angeforderten Daten vom entfernten System geladen und dem Anwender zur Verfügung gestellt.

Die Benutzung des FUSE-Frameworks bietet für die Implementierung von Dateisystemen einige Vorzüge gegenüber der Implementierung als Teil des Kernels:

- Freie Wahl der Programmiersprache. Obwohl FUSE selbst und auch viele FUSE-Dateisysteme[8] in C geschrieben sind, existiert für eine Vielzahl weiterer Sprachen FUSE-Anbindungen, die unter anderem die Nutzung von C++, Java, C#, Perl und Python ermöglichen.

- Benutzung von Standard-C-Bibliotheken (wie *libc* [18] oder *glib-2.0* [16]) und anderen Benutzermodusbibliotheken der jeweiligen Programmiersprache. Dies kann die Implementierung deutlich vereinfachen, da ein höherer Abstraktionsgrad und bessere Wiederverwendbarkeit möglich sind.

- Alle einschränkenden Eigenschaften, die für die Implementierung von Dateisystemen als Teil des Kernels im Abschnitt 2.3 auf Seite 17 beschrieben wurden, gelten für FUSE-basierte Dateisysteme in dieser Form nicht. Besonders nützlich ist der Speicherschutzmechanismus, der von FUSE-basierten Dateisystemen als Benutzermodusprogramme automatisch genutzt wird.

- Konzentration der Implementierung auf die Kernfunktionalitäten des Dateisystems, auf Grund des Framework-Charakters von FUSE. Funktionen wie das Ein- und Ausbinden (engl. *mount/unmount*) von Dateisystemen, Verwaltung des Zugriffs auf FUSE-Dateisysteme und weitere Dateisystemoptionen (zum Beispiel ausschließlicher Lesezugriff oder Aktivieren von Debug-Ausgaben), die für alle Dateisysteme notwendig sind, werden von FUSE übernommen und müssen daher nicht als Teil des Dateisystems implementiert werden.

[8] Eine Übersicht der verfügbaren FUSE-Dateisystem findet sich unter: `http://fuse.sourceforge.net/filesystems.html`.

Trotz der Tatsache, dass FUSE-basierte Dateisysteme im Benutzermodus ausgeführt werden, sind die Möglichkeiten der Implementierung von Dateisystemen denen von im Kernel implementierten Dateisystemen nahezu ebenbürtig. So beschränken sich FUSE-basierte Dateisysteme nicht lediglich auf die Abbildung vorhandener auf neue Strukturen (wie zum Beispiel im Fall von *sshfs*), sondern auch die hardwarenahe Implementierungen lokaler Dateisysteme ist möglich. Zu beachten ist, dass durch die FUSE-Architektur – also das Weiterreichen der Anfragen aus dem Kernel in den Benutzermodus und wieder zurück – zusätzlicher Verwaltungsaufwand (engl. *Overhead*) verursacht wird, so dass die Implementierung zeitkritischer Dateisysteme nur innerhalb des Kernels sinnvoll möglich ist.

FUSE-Dateisysteme sind nach dem Einbinden grundsätzlich ausschließlich für denjenigen Benutzer zugreifbar, der das Dateisystem eingebunden hat. Nicht einmal der Systemverwalter *root* hat Zugriff darauf. Dieses Konzept wird als privates Einbinden bezeichnet und kann durch den Eintrag `user_allow_other` in der FUSE-Konfigurationsdatei `/etc/fuse.conf` beeinflusst werden. Ist der Eintrag vorhanden, können FUSE-Dateisystem mit der zusätzlichen Option `-o allow_other`, was den Zugriff für alle anderen Benutzern des Systems erlaubt oder der Option `-o allow_root`, was den zusätzlichen Zugriff für den Systemverwalter erlaubt, eingebunden werden.

2.3.3 Implementierung von FUSE-Dateisystemen

Die Implementierung eines FUSE-basierten Dateisystems ähnelt der von Dateisystemen des Virtual Filesystems. Auch FUSE definiert die Schnittstellen von Dateisystemaufrufen ähnlich den Operationen von VFS-Objekten als *Callback*-Methoden [22], die von einem FUSE-Dateisystem implementiert werden können. Diese Definitionen befinden sich in der C-Header-Datei `fuse.h` innerhalb der `struct fuse_operations`-Datenstruktur. Folgende Schnittstellen sollte ein Dateisystem mindestens implementieren:

- `int (*open) (const char *, struct fuse_file_info *);`
  Diese Methode wird aufgerufen, wenn die als String übergebene Datei geöffnet werden soll. Die ebenfalls übergebene `struct fuse_file_info`-Datenstruktur liefert die Information, in welchem Modus die Datei geöffnet werden soll (Feld `flags`) und bietet Speicherplatz für ein *Filehandle* (Feld `fh`, auch *File Descriptor* genannt) und steht in allen Methoden zur Verfügung, in denen die Datei geöffnet ist.

- `int (*read) (const char *, char *, size_t, off_t, struct fuse_file_info *);`
  Aus der Datei wird eine bestimmte Anzahl von Bytes (`size_t`) von einer bestimmten Position (*Offset*, `off_t`) gelesen und in einen Puffer (`char *`) geschrieben. Als Rückgabewert wird die Anzahl der gelesenen Bytes zurückgegeben.

- `int (*write) (const char *, const char *, size_t, off_t, struct fuse_file_info *);`
  Analog zur `read()`-Operation wird in dieser Methode aus dem Puffer eine bestimmte Anzahl von Bytes gelesen und an der Offset-Position in die Datei geschrieben.

- `int (*getattr) (const char *, struct stat *);`
  In dieser Methode werden Dateiattribute (wie Dateigröße und Eigentümer) einer Datei oder eines Verzeichnisses innerhalb der `struct stat`-Datenstruktur abgelegt, die in der Datei `stat.h` definiert ist.

- `int (*readdir) (const char *, void *, fuse_fill_dir_t, off_t, struct fuse_file_info *);`
  Der erste Parameter dieser Methode enthält das angeforderte Verzeichnis. Die Aufgabe dieser Methode besteht darin, alle zu diesem Verzeichnis gehörenden Verzeichniseinträge hinzuzufügen. Dazu wird für jeden Eintrag die `filler()`-Methode aufgerufen und der entsprechende Dateiname und optional auch die entsprechenden Dateiattribute als `struct stat`-Datenstruktur als Parameter übergeben.

Mit diesen fünf Methoden ist bereits die Implementierung eines Dateisystem möglich, das lesend und schreibend auf Dateien zugreifen und Verzeichnisse anzeigen kann. FUSE definiert folgende weitere Schnittstellen: `mknod()`, `mkdir()`, `unlink()`, `rmdir()`, `rename()`, `chmod()`, `chown()`, `truncate()`, `flush()` und `release()`. Es kommen noch folgende seltener benötigte Schnittstellendefinitionen hinzu: `symlink()`, `link()`, `utime()`, `statfs()`, `fsync()`, `setxattr()`, `getxattr()`, `listxattr()`, `removexattr()`, `opendir()`, `releasedir()`, `fsyncdir()` und die FUSE-spezifischen Schnittstellen `init()` und `destroy()`, die vor dem Einbinden und nach dem Entfernen eines Dateisystems aus dem Verzeichnisbaum ausgeführt werden und so die Möglichkeit bieten, Ressourcen zu allozieren bzw. zu deallozieren. Für eine genaue Beschreibung der Schnittstellen und deren Verwendung wird auf die entsprechenden Kommentare in der Datei `fuse.h` verwiesen.

Eine Besonderheit muss im Fall eines zu signalisierenden Fehlers beachtet werden. Während Kernel-Dateisysteme der Variable `errno` den entsprechenden Fehlercode[9] zuweisen, um einen Fehler zu signalisieren, wird von FUSE-Dateisystemen erwartet, dass der negative Fehlercode direkt zurückgegeben wird. Zum Beispiel `return -ENOMEM;` anstatt `errno = ENOMEM;` im Fall eines „Out of memory“-Fehlers.

Die Implementierung jedes FUSE-basierten Dateisystems verläuft in den Grundzügen sehr ähnlich. Im Fall der Programmiersprache C muss zu Beginn die zu verwendende FUSE-Schnittstellenversion mit einer C-Präprozessordirektive [30, Chapter 10, C Preprocessor] festgelegt werden und die Header-Datei `fuse.h` eingebunden werden. Die `main()`-Methode des Dateisystem

[9] Die Fehlercodes sind in der Datei `errno-base.h` definiert.

muss im letzten Schritt, nachdem beispielsweise Ressourcen alloziert wurden, die Kontrolle an FUSE übergeben und dafür die Methode `fuse_main()` aufrufen. Als Parameter erwartet diese die Parameteranzahl und Parameterliste der `main()`-Methode. Falls das Dateisystem eigene Parameter nutzt, so sind diese nicht an `fuse_main()` weiterzureichen. Als dritter Parameter wird ein Zeiger auf die `struct fuse_operations`-Datenstruktur übergeben, in der definiert ist, welche FUSE-Operationen das Dateisystem implementiert.

Listing 2.7: Grundgerüst eines FUSE-Dateisystems

```
1  #define FUSE_USE_VERSION 22
2  #include <fuse.h>
3
4  static int myfilesystem_getattr(
5      const char *path, struct stat *stat) {...}
6  static int myfilesystem_open(
7      const char *path, struct fuse_file_info *fi) {...}
8
9  static struct fuse_operations myfilesystem_operations = {
10     .getattr    = myfilesystem_getattr,
11     .open       = myfilesystem_open,
12 };
13
14 int main(int argc, char *argv[]) {
15     return fuse_main(argc, argv, &myfilesystem_operations);
16 }
```

Die Entscheidung darüber, welche FUSE-Operationen benötigt werden und die Implementierung der jeweiligen Methodenrümpfe ist dateisystemspezifisch. Beim Übersetzen des Dateisystems mit GCC ist zu beachten, dass es gegen die FUSE-Bibliothek gelinkt werden muss (Option `-lfuse`) und dass das Symbol `_FILE_OFFSET_BITS` auf den Wert 64 gesetzt wird, damit die GNU-C-Bibliothek *libc* Aufrufe der `stat()`-Methode transparent an die `stat64()`-Methode weiterreicht [18, 14.9.2 Reading the Attributes of a File]. Nach dem Aufruf von `gcc myfs.c -o myfs -D_FILE_OFFSET_BITS=64 -lfuse` kann das Dateisystem mit dem Befehl `./myfs /tmp/mountpoint` eingebunden werden.

Kapitel 3

Anforderungen und Architektur

Dieses Kapitel verbindet die Elemente Versionskontrolle, WebDAV-Netzwerkzugriff und Dateisysteme und gibt einen Überblick über die Architektur eines Dateisystems für den transparenten Zugriff auf das Versionskontrollsystem Subversion.

3.1 Anforderungen

Folgende Anforderungen werden an ein solches Dateisystem gestellt:

1. Dateisystemgleicher Zugriff auf das Versionskontrollsystem.
   Der Anwender des Dateisystems soll keine Unterschiede zur Nutzung eines lokalen Dateisystems bemerken. Das Dateisystem muss daher alle wichtigen Dateisystemoperationen unterstützen und diese so implementieren, dass sie semantisch äquivalent mit den Operationen lokaler Dateisysteme sind.

2. Automatische Erzeugung einer neuen Version bei Änderungen.
   Die Versionierung der Änderungen soll nicht durch Anwender durchgeführt werden, sondern automatisch bei jeder Änderung an den Daten vorgenommen werden. Dies soll für den Anwender transparent und ohne zusätzlichen Aufwand geschehen.

3. Effektiver Zugriff auf alle Versionen der Daten.
   Da alle Änderungen durch das Versionskontrollsystem gespeichert werden, sollen Anwender auf alle Versionen effektiv über das Dateisystem zugreifen können.

4. Schutz vor gegenseitigem Überschreiben von Daten bei Gruppenarbeit.
   Bei der gleichzeitigen Bearbeitung von Daten sollen Vorkehrungen getroffen werden, die das gegenseitige Überschreiben von Daten verhindern, um so mögliche Datenverluste zu vermeiden und die Konsistenz der Daten sicherzustellen.

5. Netzwerkzugriff auf das Versionskontrollsystem.
   Da Versionskontrollsysteme häufig auf zentralen Servern betrieben werden, auf die eine Gruppe von Anwendern zugreift, muss auch das Dateisystem netzwerkfähig sein.

3.2 Architektur

Die Architektur eines solchen Dateisystems bestimmt sich nach diesen Anforderungen und dem Funktionsumfang des eingesetzten Versionskontrollsystems. Das im Grundlagenkapitel vorgestellte Versionskontrollsystem Subversion unterstützt alle wichtigen Dateisystemoperationen, die von einem solchen Dateisystem implementiert werden. Im Gegensatz zu dem Versionskontrollsystem CVS wird auch das Umbenennen, Verschieben und Löschen von Dateien und Verzeichnissen unterstützt, was allerdings kein Alleinstellungsmerkmal von Subversion ist.

Der entscheidende Vorteil von Subversion ist dessen Netzwerkintegration auf Basis des WebDAV-Protokolls und die damit verbundene Möglichkeit der Autoversionierung, die in Abschnitt 2.1.3 auf Seite 9 beschrieben ist. Diese Funktion erlaubt die Implementierung eines Dateisystems für den Zugriff auf allgemeine WebDAV-Ressourcen (hier auf Subversion-Repositories), während die Versionskontrolle serverseitig von Subversion übernommen wird. Durch die Verwendung von WebDAV als Kommunikationsprotokoll wird gleichzeitig die Anforderung des Netzwerkzugriffs auf das Versionskontrollsystem erfüllt.

Auf Grund dieser Eigenschaften empfiehlt sich Subversion als Versionskontrollsystem für die Implementierung eines transparenten Dateisystemzugriffs, wie folgende (vereinfachte) Architektur darstellt:

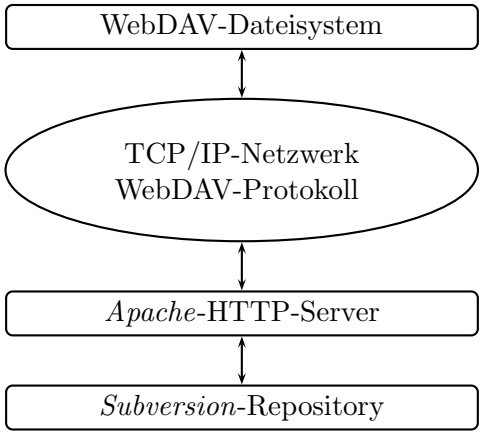

Abbildung 3.1: (Vereinfachte) Architektur eines versionierenden Dateisystems

Ein solches Dateisystem wird als versionierendes Dateisystem (engl. *Versioning Filesystem*) bezeichnet, da jede Änderung von Daten zur Erzeugung einer neuen Version führt und gleichzeitig alle Versionen zugreifbar bleiben. Die Besonderheit dieser Implementierung besteht darin, dass die Versionsverwaltung nicht als Teil des Dateisystems implementiert ist, sondern serverseitig von Subversion übernommen wird. Dies reduziert die Komplexität der Implementierung des Dateisystems und besitzt den Vorteil, dass die erprobten Versionskontrollfunktionen von Subversion an Stelle einer selbst implementierten Versionsverwaltung benutzt werden können.

Das zu implementierende Dateisystem muss neben der Anforderung des dateisystemgleichen Zugriffs auf das Versionskontrollsystem auch die Anforderung des Zugriff auf alle Versionen und die Anforderung des Schutzes vor dem gegenseitigen Überschreiben von Daten bei Gruppenarbeit erfüllen. Letzteres kann durch die Verwendung von WebDAV-Sperren (engl. *Locking*) erreicht werden, so dass Ressourcen vor dem Bearbeiten exklusiv gesperrt werden und das Überschreiben durch Dritte verhindert wird. Dies entspricht dem Lock-Modify-Unlock-Prozess, der im Grundlagenkapitel der Versionskontrolle vorgestellt wurde.

Der Zugriff auf alle Versionen der Daten entspricht dem Zugriff auf alle Revisionen eines Subversion-Repositorys. Dieser Zugriff findet über das Dateisystem so statt, dass jede Revision durch ein eigenes Verzeichnis repräsentiert wird, welches die Daten dieser Revision enthält. Diese Revisionsverzeichnisse befinden sich unterhalb eines Elternverzeichnisses, welches im Wurzelverzeichnis des Dateisystems eingeblendet wird. Die Abbildung 4.10 auf Seite 47 verdeutlicht dies.

Die Implementierung eines Dateisystems ist immer mit der Entscheidung für ein Betriebssystem verbunden. Auf Grund der Flexibilität, Modularität und der Verfügbarkeit eines Dateisystem-Frameworks ist die Entscheidung zu Gunsten des Linux-Betriebssystems ausgefallen. Für die Implementierung wird das Dateisystem-Framework FUSE eingesetzt, da die Implementierung von Dateisystemen im Benutzermodus (engl. *Userspace*), wie in Abschnitt 2.3.2 auf Seite 23 beschrieben, etliche Vorteile im Gegensatz zu einer Implementierung innerhalb des Linux-Kernels bietet.

Für den WebDAV-Zugriff wird eine C-Bibliothek benutzt, die umfangreiche Funktionen für den Zugriff auf HTTP- und WebDAV-Ressourcen bereitstellt und den kompletten Umfang des RFC 2518 [21] unterstützt.

Die Architektur eines Dateisystems für den transparenten Zugriff auf Subversion veranschaulicht folgende Grafik:

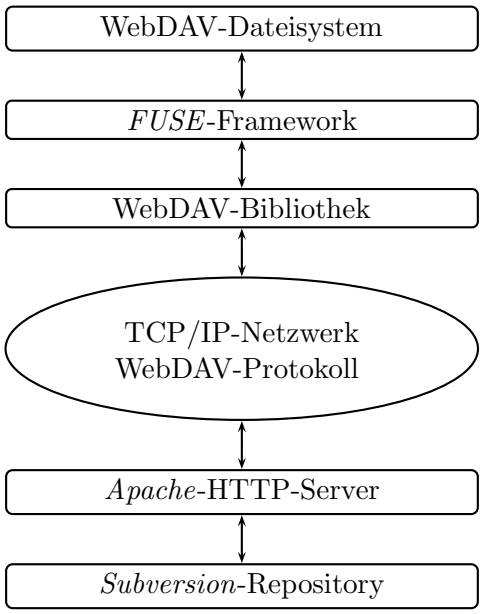

Abbildung 3.2: Architektur eines versionierenden Dateisystems

Kapitel 4

Implementierung

In diesem Kapitel wird die Implementierung eines Dateisystems für den Zugriff auf WebDAV-Ressourcen erläutert, was der obersten Schicht der im letzten Kapitel beschriebenen Architektur entspricht. Zuerst wird auf die grundlegenden Dateisystemoperationen eingegangen, die das Dateisystem zur Verfügung stellt und die Abbildung dieser Operationen auf WebDAV-Methoden. Anschließend wird die Implementierung verschiedener Strategien für das Sperren von Dateien diskutiert und auf den Zugriff aller Subversion-Revisionen eingegangen. Die Versionierung ist kein Bestandteil des Dateisystems, sondern wird von der Subversion-Autoversionierung übernommen, die im Abschnitt 2.1.3 auf Seite 9 beschrieben wurde.

Für den WebDAV-Zugriff wird die C-Bibliothek *Neon* [29] verwendet, die für Linux, andere Unix-basierte Systeme und Windows unter den Bedingungen der *GNU Lesser General Public License* [20] frei verfügbar ist. Neon stellt umfangreiche Funktionen für den Zugriff auf HTTP- und WebDAV-Ressourcen bereit und unterstützt den kompletten Umfang des RFC 2518 [21]. Die Implementierung des Dateisystems verwendet die Version 0.24.7 von Neon, da diese alle benötigten Funktionen bereitstellt und von vielen aktuellen Linux-Distributionen entsprechende Pakete bereitgestellt werden. Es kann auch die aktuellere Neon Version 0.25 verwendet werden. Alle Methoden, Datenstrukturen oder Konstanten, die mit `ne_` oder `NE_` beginnen, sind Teil der Neon-Bibliothek. Weiterhin setzt das *wdfs*-Dateisystem FUSE in der Version 2.3 und die *glib-2.0*-Bibliothek[1] und das Modul *pthread* der *libc*-Bibliothek voraus.

Das Dateisystems trägt den Namen *WebDAV Filesystem* (kurz *wdfs*) und ist in der Programmiersprache C implementiert, da die Neon-Bibliothek in C verfügbar ist und so eine nahtlose Integration möglich ist. Weiterhin spricht die hohe Ausführungsgeschwindigkeit und der geringe Speicherverbrauch für die Verwendung von C.

[1] Für Debian GNU/Linux benötigte Pakete: `libglib2.0-dev` und `libglib2.0-0`

Das Dateisystem wird als Quellcode-Paket (engl. *Tarball*, Datei `wdfs-1.0.tar.gz`[2]) bereitgestellt. Nach dem Entpacken muss es mit dem Befehl `./configure && make && make install` kompiliert und installiert werden. Für diese automatisierte Installation werden die Autotools des *GNU Build Systems* [43] benutzt, um zu überprüfen, ob alle benötigten Bibliotheken verfügbar sind und die Installation benutzerfreundlich und zuverlässig zu gestalten.

Die Implementierung ist modular aufgebaut [30, Chapter 18, Modular Programming]. Die Dateisystemoperationen in Form von FUSE-Callback-Methoden befinden sich in dem zentralen Modul `wdfs-main.c`. Das Modul `webdav.c` stellt Funktionen für den Verbindungsaufbau zu dem WebDAV-Server und für das Sperren und Entsperren von Dateien zur Verfügung. Im Modul `cache.c` befindet sich die Implementierung eines Zwischenspeichers für Dateiattribute. Mit Hilfe des Moduls `svn.c` ist der Zugriff auf alle Subversion-Revisionen eines Repositorys möglich. Jedes Modul besitzt eine Header-Datei in der exportierte Methodensignaturen und gemeinsam benutzte Datenstrukturen definiert sind.

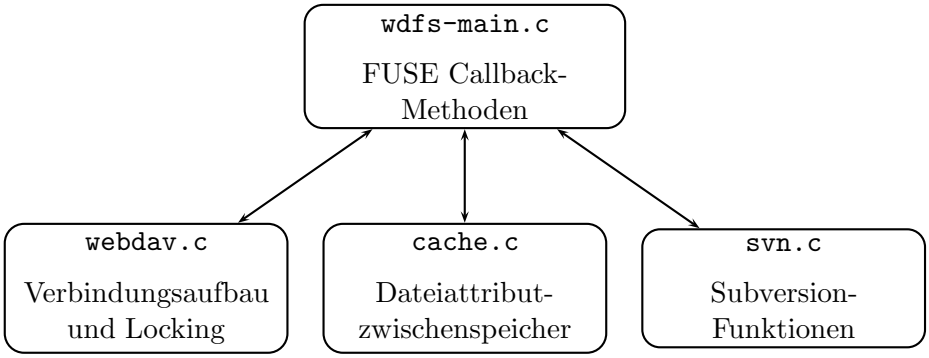

Abbildung 4.1: Module des Dateisystems *wdfs*

[2]Die Datei befindet sich auf der CD-ROM, die dieser Arbeit auf der letzten Seite hinzugefügt wurde. Alternativ kann die Datei unter dieser URL heruntergeladen werden: `http://noedler.de/projekte/wdfs/wdfs-1.0.tar.gz`

Die `main()`-Methode des Moduls `wdfs-main.c` ist für das Verarbeiten der übergebenen Parameter, den Aufbau der WebDAV-Verbindung und den Aufruf von FUSE mittels der Methode `fuse_main()` zuständig. Das Verhalten von *wdfs* lässt sich durch folgende Parameter beeinflussen:

Parameter	Beschreibung
`-v`	Versionsinformationen von *wdfs*
`-h`	Übersicht der Parameter von *wdfs* und FUSE
`-D`	Erweiterte Debug-Ausgaben von *wdfs*
`-a URI`	URI/Adresse des einzubindenden WebDAV-Servers
`-u benutzer`	Benutzername für die Authentifizierung
`-p passwort`	Passwort für die Authentifizierung
`-S`	Aktiviert Zugriff auf alle Subversion-Revisionen
`-l`	Aktiviert Locking (Sperren von geöffneten Dateien)
`-t sekunden`	Nach n Sekunden wird eine Sperre aufgehoben
`-m strategie`	Auswahl der Locking-Strategie

Abbildung 4.2: Übersicht der Parameter von *wdfs*

Zusätzlich zu diesen Parametern können FUSE-spezifische Parameter angegeben werden, die von *wdfs* an FUSE weitergereicht werden. Der Aufruf von *wdfs* für das Einbinden eines WebDAV-Servers lautet:
`wdfs mountpoint -a http://server/[verzeichnis/] [parameter]`

4.1 Dateisystemoperationen und WebDAV-Methoden

Das Dateisystem implementiert die elementaren Operationen `getattr()`, `readdir()`, `open()`, `read()`, `write()` und zusätzlich die Operationen `release()`, `truncate()`, `mknod()`, `mkdir()`, `unlink()`, `rmdir()` und `rename()`.

Die Operationen lassen sich in drei Gruppen einteilen. Die erste Gruppe besteht aus den Operationen `getattr()` und `readdir()`, da beide mit Verzeichniseinträgen (engl. *Dentries*) und Dateiattributen arbeiten. Sie hängen voneinander ab, da das Anzeigen von Einträgen (Dateien) eines Verzeichnisses ohne die entsprechenden Dateiattribute nicht möglich ist und da nur Dateiattribute von Einträgen (Dateien) angezeigt werden können, die einem Verzeichnis angehören.

Die zweite Gruppe bilden die Operationen `open()`, `read()`, `write()` und `release()`, da alle auf geöffneten Dateien und somit einer gemeinsamen Datenbasis operieren. Die restlichen Operationen sind voneinander unabhängig, da sie Funktionen ausführen, die auf keine der anderen Operationen direkten Einfluss haben. Alle hier genannten Operationen sind in der Implementie-

rung mit dem Präfix `wdfs_` versehen, welches hier aus Gründen der Lesbarkeit ausgelassen ist.

Allen Operationen ist gemeinsam, dass die angeforderte Datei (bzw. Verzeichnis; allgemein Dentry) als Parameter `char *localpath` übergeben wird. Dies ist immer ein absoluter Pfad relativ zu dem Mountpoint des Dateisystems. Wurde das Dateisystem unter `/tmp/mountpoint` eingebunden und die Datei `/tmp/mountpoint/datei` angefordert, so enthält `*localpath` den Wert `/datei`. Für den WebDAV-Zugriff muss vor diesen Pfad noch der Pfad des WebDAV-Servers hinzugefügt werden. Wurde der WebDAV-Server mit der URI `http://server/svn/repository/` eingebunden, gibt die Methode `char* get_remotepath(char *localpath)` den korrekten Pfad (hier: `/svn/repository/datei`) zurück, mit dem ein WebDAV-Zugriff auf die entfernten Daten mit den von der Neon-Bibliothek bereitgestellten Methoden möglich ist. Daher wird die Methode `get_remotepath()` von jeder Dateisystemoperation benutzt, um den lokalen Pfad entsprechend zu transformieren.

Diese Tabelle gibt einen Überblick über die implementierten Dateisystemoperation und die WebDAV-Methode, die für die jeweilige Implementierung verwendet wird. Im Folgenden werden die wichtigsten Implementierung detailliert vorgestellt.

Dateisystemoperation	WebDAV-Methode
`getattr()`	`PROPFIND`
`readdir()`	`PROPFIND`
`open()`	`GET`
`read()`	-
`write()`	-
`release()`	`PUT`
`truncate()`	`GET, PUT`
`mknod()`	`PUT`
`mkdir()`	`MKCOL`
`unlink()`	`DELETE`
`rmdir()`	`DELETE`
`rename()`	`MOVE`

Abbildung 4.3: Zuordnung der von *wdfs* implementierten Dateisystemoperationen zu WebDAV-Methoden

Gruppe 1: `getattr()` und `readdir()`

Die `getattr()`-Operation bekommt als Parameter einen Zeiger auf eine `struct stat`-Datenstruktur übergeben, welche die Attribute einer Datei oder eines Verzeichnisses repräsentiert. Diese Attribute werden bei dem Web-

DAV-Server mit Hilfe einer `PROPFIND`-Methode angefragt. Neon stellt hierfür die Methode `ne_simple_propfind()` zur Verfügung, mit der die WebDAV-Eigenschaften `DAV:resourcetype`, `DAV:getcontentlength`, `DAV:getlastmodified` und `DAV:creationdate` angefragt werden. Die Methode `ne_simple_propfind()` erwartet als Parameter einen Zeiger auf eine Funktion (d.h. eine *Callback*-Methode [22], hier `getattr_propfind_callback()`), die das Ergebnis der `PROPFIND`-Anfrage übergeben bekommt. Diese ruft wiederum die Methode `set_stat()` auf, welche die WebDAV-Antwort auswertet und die entsprechenden Werte der `struct stat`-Datenstruktur setzt, so dass die Attribute des Dateisystems angezeigt werden können. Folgende Abbildung stellt den Callback-Aufruf graphisch dar:

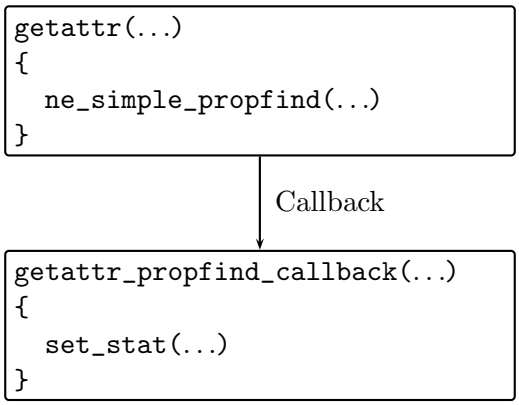

Abbildung 4.4: Callback-Aufruf in der `getattr()`-Operation (vereinfacht)

Das Setzen dieser Attribute ist mittels WebDAV gemäß RFC 2518 [21] nicht möglich. Zwar besteht die Möglichkeit mittels der `PROPPATCH`-Methode Eigenschaften zu verändern, doch die für Dateisysteme wichtigen Eigenschaften des letzten Zugriffs und der letzten Änderung besitzen keine standardisierten WebDAV-Entsprechungen. So ist die WebDAV-Eigenschaft `DAV:getlastmodified` nur lesend zugreifbar und für den Zeitpunkt des letzten Zugriffs gibt es keine standardisierte WebDAV-Eigenschaft. Daher fällt die Implementierung der `setattr()`-Operation von *wdfs* rudimentär (`return 0`) aus, so dass *wdfs* die Systemaufrufe `utime()` und `utimes()` nicht unterstützen kann.

Die Aufgabe der `readdir()`-Operation ist das Hinzufügen der Verzeichniseinträge (engl. *Dentries*) des angeforderten Verzeichnisses. Dazu bekommt sie als Parameter einen Zeiger auf die Methode `fuse_fill_dir_t` mit dem Namen `filler` übergeben, die für jeden Eintrag (jede Datei) des Verzeichnisses mit dem entsprechenden Namen der Datei als Parameter aufgerufen werden muss. Die `filler()`-Methode erwartet als weitere Parameter einen Puffer, dem der Verzeichniseintrag hinzugefügt werden soll und optional auch die Attribute der Datei.

Die Informationen welche Dateien zu dem Verzeichnis gehören, wird mit Hilfe der `ne_simple_propfind()`-Methode bei dem WebDAV-Server per `PROPFIND`-Methode angefragt. Für jede Datei der WebDAV-Antwort (allgemein für alle Mitglieder der angefragten Kollektion) wird von dieser Funktion deren Callback-Methode `readdir_propfind_callback()` aufgerufen, die wiederum die Methode `set_stat()` und die Methode `filler()` mit den entsprechenden Parametern aufruft, um den jeweiligen Verzeichniseintrag zu dem aktuellen Verzeichnis hinzuzufügen. Folgende Abbildung stellt die Callback-Aufrufe grafisch dar:

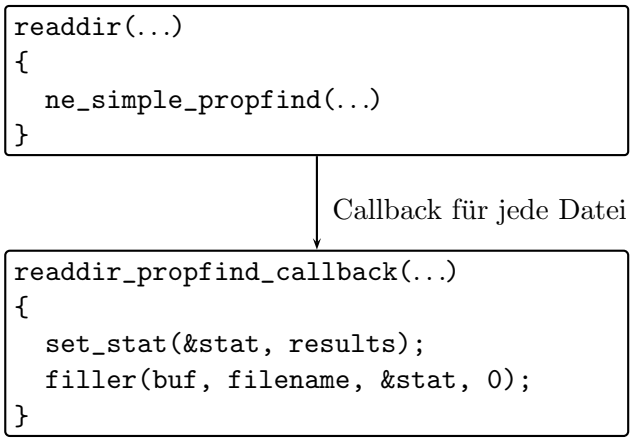

Abbildung 4.5: Callback-Aufrufe in der `readdir()`-Operation (vereinfacht)

Da die `filler()`-Methode nicht direkt im Rumpf der `readdir()`-Operation aufgerufen werden kann, wurde die `struct dir_item`-Datenstruktur in der Datei `wdfs-main.h` definiert. Sie enthält einen Zeiger auf die `filler()`-Methode, den Puffer der `filler()`-Methode und den Pfadnamen des angefragten Verzeichnisses. Diese Datenstruktur wird der `ne_simple_propfind()`-Methode übergeben, so dass diese innerhalb deren Callback-Methode `readdir_propfind_callback()` zugreifbar ist und dort wie beschrieben die `filler()`-Methode aufgerufen werden kann.

Gruppe 2: `open()`, `read()`, `write()` und `release()`

Bei der Implementierung von *wdfs* wird von der Annahme ausgegangen, dass Dateien, die geöffnet werden, auch gelesen oder geschrieben werden. Daher besteht die Aufgabe der `open()`-Operation darin, die angefragte Datei mit Hilfe der Neon-Methode `ne_get()` von dem WebDAV-Server zu laden und in ein Filehandle zu schreiben. Die Operationen `read()` und `write()` benutzen die Daten des Filehandles ohne weitere Kommunikation mit dem Server. Erst beim Schließen der Datei wird die `release()`-Operation ausgeführt und die Datei, falls sich deren Inhalt geändert hat, wieder zu dem Server

gesandt. Durch diese Strategie konnten die Operationen für das Lesen und Schreiben sehr effektiv implementiert werden. Der Nachteil dieser Strategie besteht darin, dass auch Dateien von dem Server geladen werden, die geöffnet und wieder geschlossen werden, ohne eine Lese- oder Schreiboperation auszuführen.

In der `open()`-Operation werden die Daten von der `ne_get()`-Methode in ein zuvor mit der Methode `get_filehandle()` geöffnetes Filehandle geschrieben [28, Appendix B.1]. Das Filehandle wird innerhalb der `open()`-Operation in der selbstdefinierten `struct open_file`-Datenstruktur gespeichert, die ein weiteres Feld enthält, das signalisiert, ob die Daten des Filehandles verändert wurden (`bool_t modified`). Um diese Datenstruktur auch in den Operationen für das Lesen und Schreiben sowie beim Schließen der Datei verfügbar zu machen, wird ein Zeiger darauf in der `struct fuse_file_info`-Datenstruktur gespeichert, da diese auch in den Operationen `read()`, `write()` und `release()` zur Verfügung steht.

Die Implementierung der `read()`- und `write()`-Operationen ist daher sehr übersichtlich und doch vollständig, da sie durch die Benutzung der Methoden `pread()` bzw. `pwrite()` alle Modi für das Lesen und Schreiben von Dateien (wie `O_APPEND` oder `O_RDWR`) abdeckt.

Listing 4.1: *wdfs* `read()`-Operation (vereinfacht)

```
1 struct open_file *file = (struct open_file*)fi->fh;
2 pread(file->fh, buf, size, offset);
```

Listing 4.2: *wdfs* `write()`-Operation (vereinfacht)

```
1 struct open_file *file = (struct open_file*)fi->fh;
2 pwrite(file->fh, buf, size, offset);
3 file->modified = true;
```

In der `release()`-Operation werden die Daten, falls sie sich geändert haben (`file->modified == true`), mit Hilfe der `ne_put()`-Methode zu dem WebDAV-Server gesandt, das Filehandle wieder geschlossen und der alloziert Speicher für die `struct open_file`-Datenstruktur wieder freigegeben.

Die folgende Grafik stellt als Sequenzdiagramm die Dateisystemaufrufe aus Sicht eines Anwenders dar, der mit einer Applikation eine Datei öffnet und bearbeitet. Die Aufrufe des Anwenders werden von der Applikation an *wdfs* weitergereicht, welches die entsprechenden WebDAV-Methoden benutzt, um die angeforderten Daten von einem WebDAV-Server zu laden.

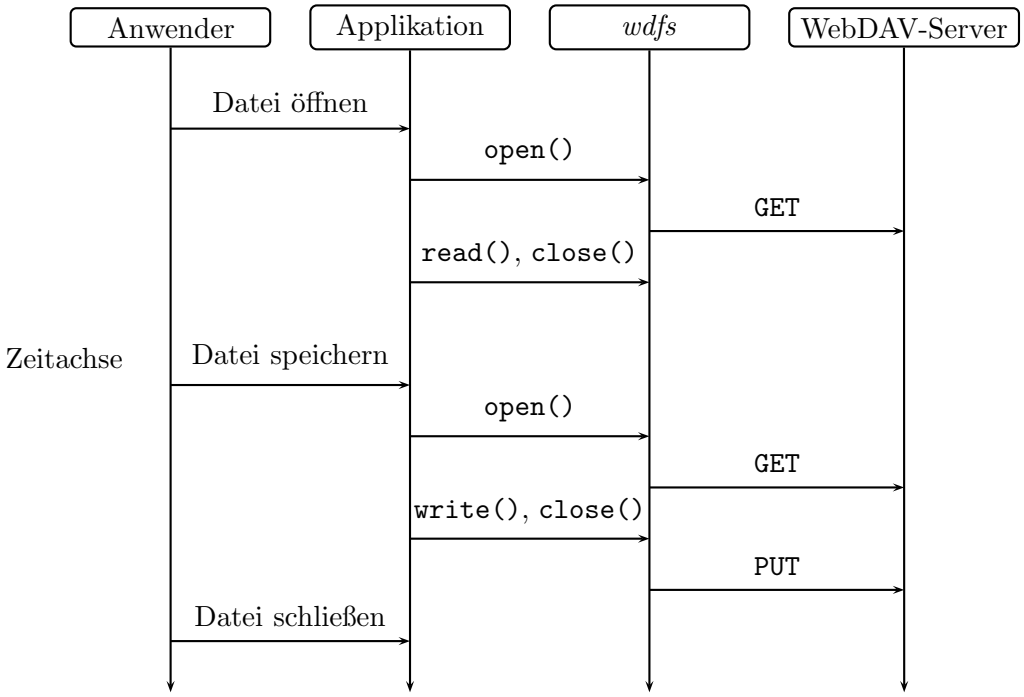

Abbildung 4.6: Sequenzdiagramm: Aufrufe eines Anwenders, einer Applikation und WebDAV-Methoden im Zusammenspiel

Gruppe 3

Die Implementierung der Operationen `mknod()`, `mkdir()`, `unlink()`, `rmdir()` und `rename()` besteht hauptsächlich darin, für die in der Tabelle auf Seite 34 aufgeführte WebDAV-Methode, die entsprechende Neon-Methode aufzurufen. Daher wird darauf nicht näher eingegangen.

Da für die `truncate()`-Operationen, deren Aufgabe das Verändern der Größe einer Datei ist, keine direkte WebDAV-Entsprechung existiert, wird deren Quelltext genauer besprochen.

Listing 4.3: *wdfs* `truncate()`-Operation (vereinfacht)

```
1  static int truncate(const char *localpath, off_t size) {
2      char *remotepath = get_remotepath(localpath);
3
4      char buffer[size];
5      memset(buffer, 0, size);
6
7      int ret, fh_in  = get_filehandle();
8      int      fh_out = get_filehandle();
9
10     if (size != 0) {
11         if (ne_get(session, remotepath, fh_in)) {
12             printf("GET error: %s\n", ne_get_error(session));
13             return -ENOENT;
14         }
15         ret = pread(fh_in, buffer, size, 0);
16         if (ret < 0) {
17             printf("pread() error: %d\n", ret);
18             return -EIO;
19         }
20     }
21
22     ret = pwrite(fh_out, buffer, size, 0);
23     if (ret < 0) {
24         printf("pwrite() error: %d\n", ret);
25         return -EIO;
26     }
27     if (ne_put(session, remotepath, fh_out)) {
28         printf("PUT error: %s\n", ne_get_error(session));
29         return -EIO;
30     }
31
32     return 0;
33 }
```

Die neue Dateigröße wird als Parameter `off_t size` übergeben und ein entsprechend großer String-Puffer (`char buffer[size]`, Zeile 4) angelegt und mit Null-Bytes gefüllt. Soll die Dateigröße auf null Bytes geändert werden, werden mittels `ne_put()` null Bytes in die Datei geschrieben, so dass die Datei anschließend die gewünschte Größe aufweist (Zeilen 22 bis 30). Bei anderen Dateigrößen außer null Bytes wird die Datei mittels `ne_get()` von dem Server geladen und in dem String-Puffer abgelegt (Zeilen 10 bis 20). Soll die Datei verkleinert werden, so wird aus dem Puffer nur die gewünschte Anzahl an Bytes gelesen und mittels `ne_put()` in die Datei geschrieben. Soll die Datei vergrößert werden entsprechend viele Null-Bytes am Dateiende hinzugefügt und mittels `ne_put()` in die Datei geschrieben, so dass die Datei immer die gewünschte Größe in Bytes besitzt. Da die Methoden `ne_get()` und `ne_put()` nicht direkt mit String-Puffern zusammenarbeiten, wurden Filehandles benutzt.

4.2 Zwischenspeichern von Dateiattributen

Eine `PROPFIND`-Anfrage einer Kollektion (Verzeichnis) liefert mit einer `207 Multi-Status`-Antwort die angefragten Eigenschaften aller Mitglieder der Kollektion, was deutlich schneller ist, als dies für jedes Mitglied (jede Datei) einzeln zu tun. Dies wird bei der Implementierung der `readdir()`-Operation genutzt, um nicht nur den benötigten Dateinamen anzufragen, sondern alle Dateiattribute die von der `getattr()`-Operation benötigt werden.

Das Ergebnis dieser Anfrage wird als `struct stat`-Datenstruktur in einem Zwischenspeicher (engl. *Cache*) abgelegt, so dass die `getattr()`-Operation darauf zurückgreifen kann und nicht für jede Datei einzelne `PROPFIND`-Anfragen an den Server senden muss. Da zu jeder `struct stat`-Datenstruktur ein eindeutiger Pfad gehört, bietet es sich an, diese in einem Hash [26, Chapter 6.4, Hashing] zu speichern und den Pfadnamen als Hash-Schlüssel zu verwenden und die entsprechende `struct stat`-Datenstruktur als Hash-Wert. Die Verwendung eines Hashs als Datenstruktur und die Eindeutigkeit der Pfadnamen erlauben schnellen $O(1)$-Zugriff auf die jeweiligen Dateiattribute. Das Zwischenspeichern der Attribute beschleunigt den Zugriff auf WebDAV-Ressourcen mit *wdfs* deutlich.

Als Hash-Wert wird an Stelle der `struct stat`-Datenstruktur die selbstdefinierte `struct cache_item`-Datenstruktur verwendet, um für jeden Hash-Eintrag zusätzlich festzuhalten, wann er zu dem Hash hinzugefügt wurde.

Listing 4.4: *wdfs* `struct cache_item`-Datenstruktur

```
1 struct cache_item {
2     struct stat stat;
3     time_t timeout;
4 };
```

Der `timeout`-Wert wird beim Hinzufügen zu dem Hash gesetzt. So wird es möglich den Hash regelmäßig nach veralteten Einträgen zu durchsuchen und diese zu entfernen. Die Lebensdauer eines Hash-Eintrags wird über die C-Präprozessordirektive `CACHE_ITEM_TIMEOUT` gesteuert und besitzt einen Standardwert von 20 Sekunden. Zu dem Zeitpunkt der Initialisierung des Dateisystems wird ein zweiter Thread [32, Chapter 12, POSIX Threads] [28, Chapter 4, Threads] gestartet, der im Abstand von 20 Sekunden den Hash nach veralteten Einträgen durchsucht und diese entfernt, damit keine möglicherweise veralteten Dateiattribute benutzt werden.

Die Implementierung dieses Zwischenspeichers befindet sich in der Datei `cache.c` und stellt folgende Methoden bereit:

1. `void cache_initialize();`
   Erstellt das Hash-Objekt und startet den Thread, der regelmäßig veraltete Einträge des Hashs entfernt.

2. `void cache_destroy();`
   Beendet den zweiten Thread, entfernt alle Einträge des Hashs und das Hash-Objekt selbst.

3. `void cache_add_item(`
   `struct stat *stat, const char *remotepath);`
   Fügt die übergebene `struct stat`-Datenstruktur gekapselt in einer `struct cache_item`-Datenstruktur zu dem Hash hinzu und setzt den entsprechenden `timeout`-Wert. Als Hash-Schlüssel wird der eindeutige Pfad `remotepath` benutzt.

4. `void cache_delete_item(const char *remotepath);`
   Entfernt den zu dem Pfad gehörenden Hash-Eintrag.

5. `int cache_get_item(`
   `struct stat *stat, const char *remotepath);`
   Setzt den `stat`-Zeiger auf den zu dem Pfad gehörenden Hash-Eintrag, falls dieser im Hash vorhanden ist und gibt 0 zurück oder -1, falls kein passender Hash-Eintrag gefunden wurde.

4.3 Strategien für das Sperren von Dateien

wdfs implementiert verschiedene Strategien für das Sperren von Dateien einer WebDAV-Ressource. Unter Verwendung der WebDAV-Methoden `LOCK` und `UNLOCK` bzw. der entsprechenden Neon-Methoden `ne_lock()` und `ne_unlock()` sperrt *wdfs* Dateien vor dem schreibenden Zugriff Dritter, wenn es mit dem Parameter `-l` (für engl. *Locking*) gestartet wurde. Bei den Sperren handelt es sich um *Write Locks*, welche die einzigen im RFC 2518 [21] definierten Sperren sind. Solche Sperren verhindern das Bearbeiten einer Ressource durch Dritte, während der Lesezugriff auf gesperrte Ressourcen nicht eingeschränkt wird.

Bevor auf die einzelnen Strategien eingegangen werden kann, muss auf unterschiedliche Sichtweisen von Anwendern und Applikationen auf Dateioperationen eingegangen werden. Während Anwender Dateien öffnen, lesen und bearbeiten, speichern und schließen, lässt sich diese Sicht nicht auf die tatsächlich stattfindenden Dateioperationen übertragen. Öffnet ein Anwender eine Datei, so muss die Applikation diese öffnen, lesen und schließen. Ändert der Anwender die Datei und speichert diese ab, so muss die Applikation diese öffnen, schreiben und schließen. Die tatsächlich stattfindenden Dateioperationen unterscheiden sich also von denen, die der Anwender wahrnimmt. Um sowohl aus Sicht der Anwender als auch Sicht der Applikationen sinnvolles Sperren von Dateien zu ermöglichen, implementiert *wdfs* mehrere Strategien.

Das Ziel der Strategien ist die Datenkonsistenz durch serialisierte Schreibzugriffe sicherzustellen und das Überschreiben von Daten durch Dritte zu verhindern. Jede der folgenden Strategien für das Sperren von Dateien kann bei dem Start von *wdfs* mit dem Parameter `-m strategie_nr` ausgewählt werden:

1. Die erste Strategie sperrt eine Datei, sobald diese geöffnet wird und entsperrt diese wieder, wenn sie geschlossen wird. Diese Tabelle verdeutlicht diese Strategie.

Dateioperation	*wdfs*-Operation	WebDAV-Methode
`open()`	`open()`	`LOCK, GET`
`read()`	`read()`	-
`close()`	`release()`	`UNLOCK`
`open()`	`open()`	`LOCK, GET`
`write()`	`write()`	-
`close()`	`release()`	`PUT, UNLOCK`

   Dieses Vorgehen schützt Daten vor gleichzeitigem Schreibzugriff, so dass nicht mehrere Anwender *gleichzeitig* eine Ressource verändern können. So kann die Datenkonsistenz sichergestellt werden, doch schützt die Strategie nicht vor dem Überschreiben der Daten durch Dritte. Öffnen zwei Anwender die selbe Datei und nehmen Änderungen daran

vor, so enthält die Datei nur die Änderungen des Anwenders, der sie zuletzt geschrieben hat. Durch die Autoversionierung von Subversion bedeutet Überschreiben von Daten lediglich, dass diese nicht mehr Teil der aktuellen Revision sind und durch die Möglichkeit des Zugriffs auf vorherige Revisionen nicht verloren sind.

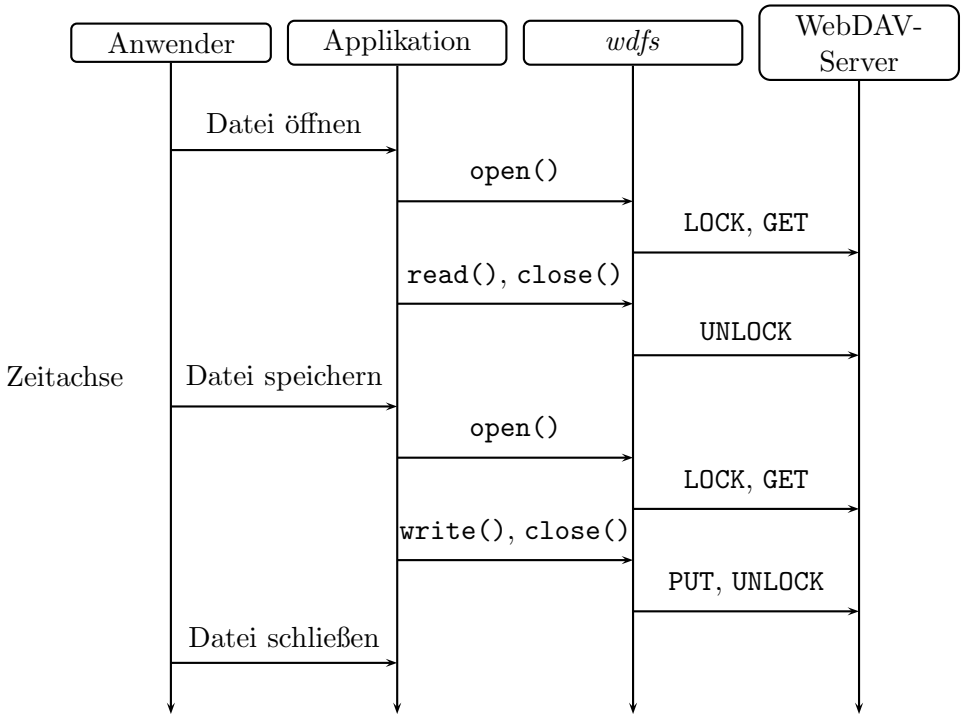

Abbildung 4.7: Strategie 1 als Sequenzdiagramm

2. Die zweite Strategie orientiert sich an der Sicht des Anwenders und sperrt eine Datei beim Öffnen und entsperrt diese erst wieder, nachdem sie geschrieben und geschlossen wurde. So ist es für Dritte nicht möglich die gesperrte Datei zu ändern, während sie bearbeitet wird, wie diese Tabelle zeigt.

Dateioperation	*wdfs*-Operation	WebDAV-Methode
`open()`	`open()`	`LOCK, GET`
`read()`	`read()`	-
`close()`	`release()`	-
`open()`	`open()`	`GET`
`write()`	`write()`	-
`close()`	`release()`	`PUT, UNLOCK`

RFC 2518 [21] empfiehlt für das Bearbeiten von Ressourcen diese Sequenz von WebDAV-Methoden, um sowohl die Datenkonsistenz als auch das Überschreiben der Daten durch Dritte zu verhindern.

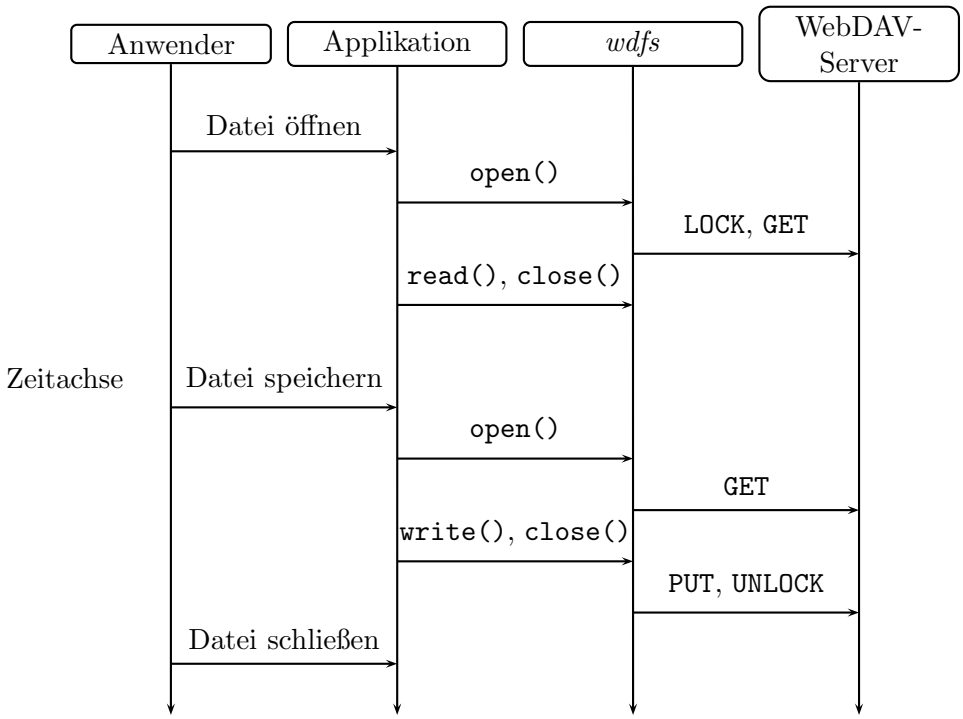

Abbildung 4.8: Strategie 2 als Sequenzdiagramm

Bei dieser Strategie ist zu beachten, dass die Sperre bei dem Schließen der Datei nach einem Schreibvorgang entfernt wird. Dies bedeutet, dass ein weiterer Schreibvorgang (ohne erneutes Öffnen der Datei) nicht mehr durch die Sperre geschützt ist und dass die Sperre ohne Schreibvorgang nicht entfernt wird. Diese Einschränkungen ergeben sich daraus, dass ein Dateisystem nicht wissen kann, wann eine Datei aus Sicht des Anwenders geschlossen wird. Daher wird davon ausgegangen, dass nach dem Schreiben und Schließen einer Datei die Sperre entfernt werden kann.

3. Sollen mehrere Schreibvorgänge einer Datei durchführen, die durch eine Sperre geschützt sind, so empfiehlt sich die dritte Strategie, die eine Sperre beim Öffnen einer Datei setzt und erst beim Beenden des Dateisystems entfernt. Einmal gesperrte Dateien bleiben somit dauerhaft für den Anwender exklusiv bearbeitbar, wie folgende Tabelle verdeutlicht.

Dateioperation	*wdfs*-Operation	WebDAV-Methode
`open()`	`open()`	`LOCK, GET`
`read()`	`read()`	-
`close()`	`release()`	-
`open()`	`open()`	`GET`
`write()`	`write()`	-
`close()`	`release()`	`PUT`
...	...	...
`unmount()`	`destroy()`	`UNLOCK`

Einmal gesperrte Dateien können beliebig oft gelesen und geschrieben werden und die Sperre wird erst beim Beenden des Dateisystems entfernt.

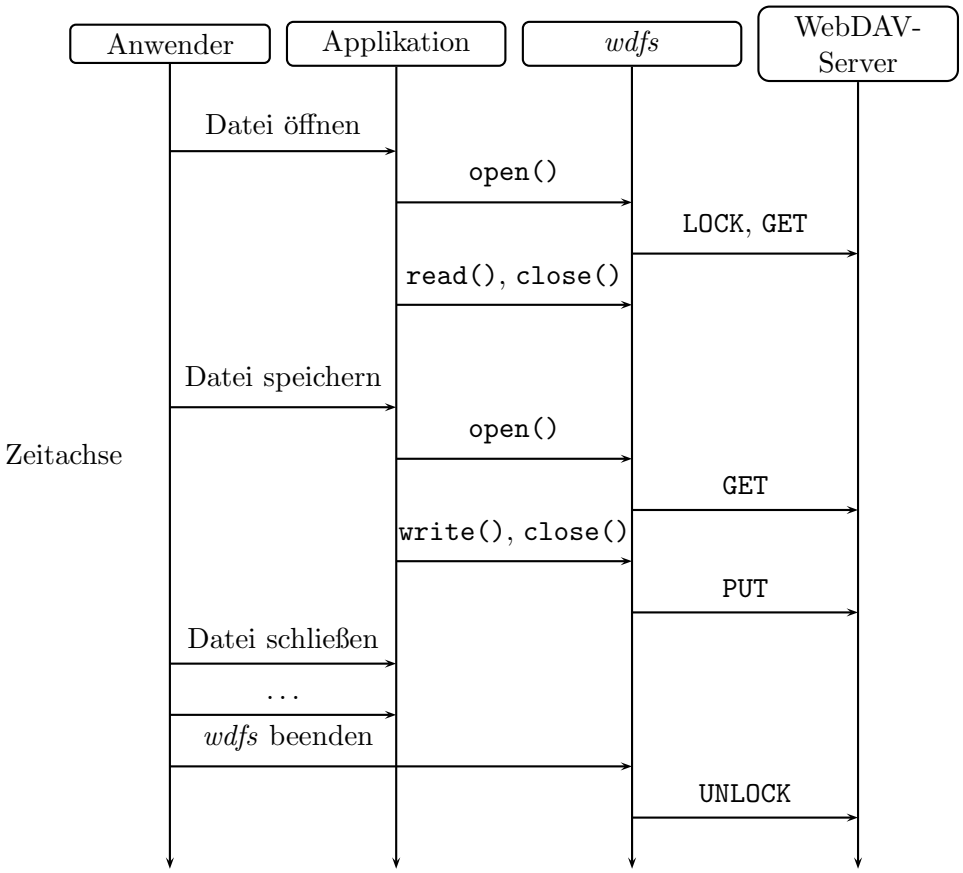

Abbildung 4.9: Strategie 3 als Sequenzdiagramm

Für jede Strategie ist zu beachten, dass Sperren nur eine bestimmte Lebensdauer besitzt. Diese kann bei dem Start von *wdfs* mit dem Parameter `-t sekunden` (engl. *Timeout*) angegeben werden. Endet die Lebensdauer, so wird die Sperre automatisch vom WebDAV-Server entfernt und die entsprechende Ressource kann erneut gesperrt und bearbeitet werden. Der Wert -1 bedeutet eine unendliche Lebensdauer einer Sperre. Wurde der Parameter `-t sekunden` nicht angegeben, so wird von *wdfs* eine Lebensdauer von 5 Minuten vorgegeben.

Ein Aufruf von *wdfs* mit der dritten Strategie könnte `wdfs mountpoint -a http://server/[verzeichnis/] -l -m 3 -t -1` lauten. In diesem Beispiel besitzen Sperren eine unendliche Lebensdauer und werden erst beim Beenden von *wdfs* wieder entfernt.

Zu beachten ist, dass Sperren keinen absoluten Schutz bieten, da sie auch von Dritten aufgehoben werden können. Der für das Entsperren notwendige Lock-Token ist Teil der `DAV:lockdiscovery`-Eigenschaft der gesperrten Ressource und kann von Dritten ausgelesen werden, um mit der `UNLOCK`-Methode die Sperre aufzuheben. Dies wird als Stehlen einer Sperre bezeichnet.

Die Sperren sind anwender- und nicht applikationsbasiert implementiert. Daher kann ein Anwender durch das Öffnen einer Datei mit Applikation A die Datei sperren und sie trotzdem mit Applikation B bearbeiten. Die Sperren schützen nur vor dem schreibenden Zugriff anderer Anwender und nicht vor anderen Applikationen des selben Anwenders. Der Vorteil liegt darin, dass der Anwender die Kontrolle über die Sperren behält, da diese nicht an die jeweilige Applikation gebunden sind.

Für die Implementierung wird ein *Lockstore*-Objekt der Neon-Bibliothek (`ne_lock_store`) benutzt, dem jede gesetzte Sperre hinzugefügt wird. Fordert nun eine (zweite) Applikation des selben Anwenders eine Sperre für eine Datei an, so wird zuerst geprüft, ob sich im Lockstore eine entsprechende Sperre befindet und falls dies der Fall ist, der Applikation signalisiert, dass die Datei bereits (für diesen Anwender) gesperrt ist und bearbeitet werden kann.

4.4 Zugriff auf alle Subversion-Revisionen

Wurde *wdfs* mit dem Parameter `-S` gestartet, so wird der Zugriff auf alle Revisionen eines Subversion-Repositorys ermöglicht. Der Zugriff erfolgt über ein virtuelles Verzeichnis, welches in der `readdir()`-Operation mit einem zusätzlichen Aufruf der `filler()`-Methode zu dem Wurzelverzeichnis von *wdfs* hinzugefügt wird. Das virtuelle Verzeichnis trägt standardmäßig den Namen `0-all-revisions/`, der in der Datei `svn.c` durch die Variable `svn_basedir` festgelegt ist.

Innerhalb des virtuellen Verzeichnisses befindet sich für jede Revision des Repositorys ein Verzeichnis, das nach der Revisionsnummer benannt ist und in dem sich alle zu dieser Revision gehörenden Dateien und Verzeichnisse befinden – d.h. der Zustand des Repositorys zu dem entsprechenden Zeitpunkt.

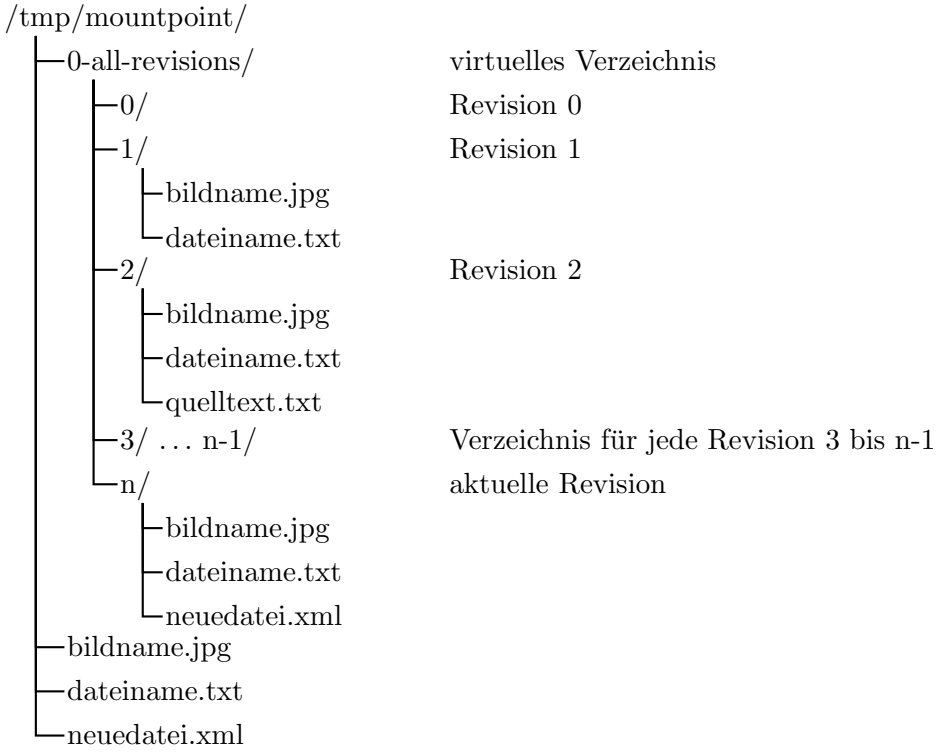

Abbildung 4.10: Verzeichnisbaum von *wdfs* für den Zugriff auf alle Subversion-Revisionen

Für jedes Revisionsverzeichnis wird eine `PROPFIND`-Anfrage an den WebDAV-Server gestellt, um den Erstellungszeitpunkt der jeweiligen Revision zu erfahren. Dadurch wird dem Anwender die Zuordnung einer Revisionsnummer zu einem Datum ermöglicht und gezieltes Zugreifen auf die gewünschte Revision möglich. Die Benutzung des virtuellen Verzeichnisses wird bei

einem Subversion-Repository mit vielen Revisionen allerdings langsam, da der WebDAV-Server für jedes Verzeichnis eine `PROPFIND`-Anfrage beantworten muss. Um die Zugriffgeschwindigkeit auf die Revisionsverzeichnisse zu verbessern, implementiert *wdfs* ein zweistufiges Modell, welches immer eine feste Anzahl von Revisionsverzeichnissen in einem eigenen Verzeichnis gruppiert und so die Anzahl der maximal benötigten `PROPFIND`-Anfragen konstant hält:

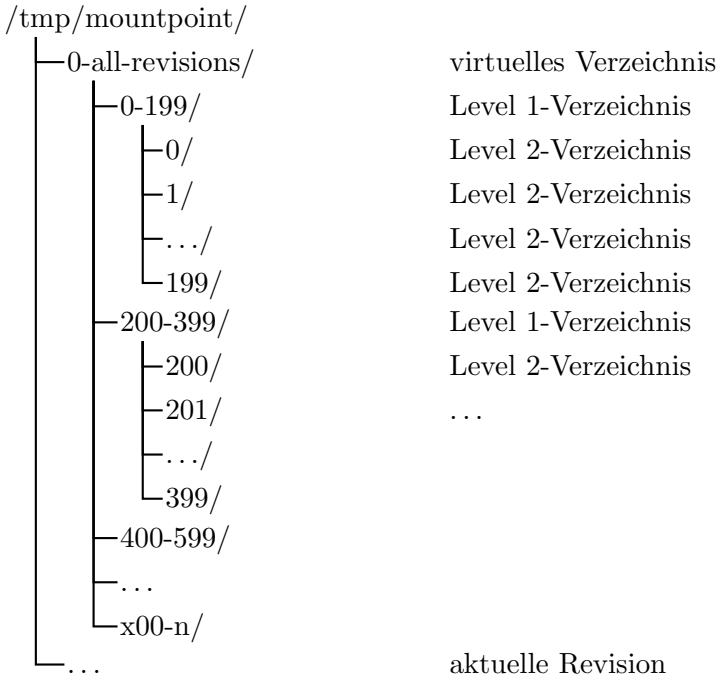

Abbildung 4.11: Verzeichnisbaum von *wdfs* für den Zugriff auf alle Subversion-Revisionen über das virtuelle Verzeichnis, Level 1- und Level 2-Verzeichnisse

Dabei werden die gruppierenden Verzeichnisse als Level 1-Verzeichnisse und die Revisionsverzeichnisse, die die Daten der jeweiligen Revision enthalten, als Level 2-Verzeichnisse bezeichnet. Die Anzahl der Level 2-Verzeichnisse pro Level 1-Verzeichnis kann in der Datei `svn.c` angepasst werden.

Für die Implementierung ist es notwendig die aktuelle Revisionsnummer zu erfahren, um die entsprechende Anzahl Level 1- und Level 2-Verzeichnisse zu erstellen. Die Methode `svn_get_latest_revision()` führt dafür eine `PROPFIND`-Anfrage der `DAV:checked-in`-Eigenschaft für die URI `/!svn/vcc/default` durch. Alle URIs, die mit `/!svn` beginnen, sind besondere URIs, die das WebDAV-Subversion-Modul zur Verfügung stellt.

Innerhalb der `readdir()`-Operation muss anhand des aktuellen Pfades (`localpath`) entschieden werden, ob das angefragte Verzeichnis

- das Wurzelverzeichnis des Dateisystems ist, so dass das virtuelle Verzeichnis `0-all-revisions/` dem Verzeichnisbaum hinzugefügt werden muss oder
- das virtuelle Verzeichnis `0-all-revisions/` ist, so dass diesem die Level 1-Verzeichnisse hinzugefügt werden müssen oder
- ein Level 1-Verzeichnis ist, so dass diesem alle entsprechenden Level 2-Verzeichnisse hinzugefügt werden müssen oder
- ein Level 2-Verzeichnis ist, so dass die Dateien und Verzeichnisse der entsprechenden Revision hinzugefügt werden müssen.

Für jede dieser Aufgaben werden entsprechende Methoden des Moduls `svn.c` benutzt. So fügt die Methode `svn_add_level1_directories()` mit Hilfe der `filler()`-Methode alle Level 1-Verzeichnisse unterhalb des virtuellen Verzeichnisses hinzu und die Methode `svn_add_level2_directories()` fügt unterhalb jedes Level 1-Verzeichnisses die jeweiligen Level 2-Verzeichnisse hinzu. Dafür sind teilweise Stringoperationen notwendig, die in C sehr aufwändig und wenig elegant zu realisieren sind. Beispielsweise muss die Methode `svn_add_level2_directories()` einen String `/0-all-revisions/200-399/` so verarbeiten, dass zwei Integer-Variablen die Werte 200 und 399 benutzen können, um in einer `for`-Schleife die Level 2-Verzeichnisse hinzuzufügen.

Der Zugriff auf den Inhalt der jeweiligen Revision ist über URIs der Form `/!svn/bc/revisionsnummer/` möglich. Analog zu der Methode `get_remotepath()` existiert daher die Methode `svn_get_remotepath()`, die einen lokalen Pfad der Form `/0-all-revisions/200-399/222/datei` in einen Pfad für den WebDAV-Zugriff der Form `/svn/repository/!svn/bc/222/datei` transformiert.

Der Zugriff auf die Daten unterhalb des virtuellen Verzeichnisses ist nur lesend möglich, daher sind lediglich die Operationen `getattr()`, `readdir()`, `open()`, `read()` und `release()` erlaubt. Jede andere Operation liefert einen `EROFS`-Fehler („Read-only file system“) zurück.

4.5 Einschränkungen des Dateisystems

Die Einschränkungen von *wdfs* lassen sich in zwei Gruppen einteilen. Die Einschränkungen der ersten Gruppe sind von der Implementierung abhängig und können daher behoben werden. Diese Einschränkungen sind vorhanden, da die Implementierung als Teil dieser Bachelorarbeit erstellt wurde und mit der Implementierung nur die Möglichkeit des Dateisystemzugriffs auf

das Versionskontrollsystem Subversion gezeigt werden sollte. Die Einschränkungen der zweiten Gruppe ergeben sich aus den eingesetzten Technologien und sind daher nicht ohne weiteres zu beheben.
Gruppe 1, implementierungsbedingte Einschränkungen:

- Die aktuelle Implementierung bietet keine Unterstützung sicherer SSL-verschlüsselter (engl. *Secure Sockets Layer*) Verbindungen über das `https://`-Schema. Da HTTP- bzw. WebDAV-Kommunikation grundsätzlich unverschlüsselt abläuft, können Dritte die Kommunikation mitlesen und manipulieren. Daher ist eine verschlüsselte Verbindung besonders dann wichtig, wenn die Kommunikation über ungesicherte Netze stattfindet. SSL-verschlüsselte Verbindungen können die geforderte Authentizität der Kommunikation sicherstellen.

- Die aktuelle Implementierung bietet keine Multithread-Unterstützung. *wdfs* wird im FUSE-Singlethread-Modus (Parameter `-s`) gestartet, so dass das Dateisystem immer nur eine Anfrage gleichzeitig bearbeiten muss. Diese Einschränkung hat lediglich Auswirkungen auf die Ausführungsgeschwindigkeit des Dateisystems bei dem gleichzeitigen Zugriff mehrerer Anwender. Für die Implementierung einer Multithread-Unterstützung müssten kritische Regionen durch Semaphoren vor konkurrierendem Zugriff geschützt werden [28, Chapter 4.4] und für jeden Thread müsste eine eigene WebDAV-Verbindung hergestellt werden.

- Da die Subversion-Revisionsnummer als Integer implementiert ist, kann *wdfs* nur auf Repositories mit maximal $2^{31} - 1$ (i386 Architektur) Revisionen Zugriff ermöglichen. Diese Einschränkung würde sich durch die Verwendung eines größeren Datentyps wie Long Integer entschärfen lassen.

Gruppe 2, technologiebedingte Einschränkungen:

- Wie auf Seite 35 beschrieben, unterstützt *wdfs* das Setzen der Dateiattribute „letzter Zugriff“ und „letzte Änderung“ mittels der Systemaufrufe `utime()` und `utimes()` nicht. Dies ist bedingt durch die WebDAV-Spezifikation [21], die für diese Attribute keine WebDAV-Standardeigenschaften vorsieht.

- Die aktuelle Implementierung des Dateisystems unterstützt nur UTF-8-codierte Datei- und Verzeichnisnamen, da Subversion intern alle Daten UTF-8-codiert speichert. Da viele Applikationen UTF-8-fähig sind und die ersten 128 Zeichen der UTF-8-Codierung mit denen der ASCII-Codierung übereinstimmen, ist die Benutzbarkeit des Dateisystem in den meisten Fällen gegeben.

4.6 Qualitätssicherung

4.6.1 Kompatibilitätstest

Die Kompatibilität des Dateisystems *wdfs* wurde mit folgenden WebDAV-Servern erfolgreich getestet:

- Apache-HTTP-Server v2.0.54 in der auf Seite 7 vorgestellten Konfiguration mit WebDAV-Subversion-Modul,
- Apache-HTTP-Server v2.0.54 ohne WebDAV-Subversion-Modul, sondern mit Apache-WebDAV-Modulen für WebDAV-Zugriff auf ein Verzeichnis an Stelle eines Subversion-Repositorys,
- Apache Tomcat v5.5.9 [3] Java Servlet/JSP-Server mit aktivierter WebDAV-Unterstützung,
- Jakarta Slide v2.1 [25] Content Management Framework mit WebDAV-Unterstützung,
- W3C's Java Server „Jigsaw WebDAV Package" [39] basierend auf Jigsaw v2.1.2.

4.6.2 Funktionale Tests

Die Implementierung wurde Tests mit dem Linux-Systemprogramm `diff` unterzogen, welches Unterschiede zwischen Dateien feststellen kann. Es handelt sich hierbei um Blackbox-Tests der Funktion des Dateisystems und nicht um Testfälle für einzelne Methoden der Implementierung. Für die Tests mit dem Programm `diff` wurden Text- und Binärdateien von wenigen Kilobyte bis zu mehreren Megabyte von einem lokalen Laufwerk in ein von *wdfs* verwaltetes Verzeichnis kopiert. Die Kopien wurden mit den Originaldateien verglichen (`diff /home/user/original /tmp/wdfs/kopie`), um die Integrität der kopierten Dateien und damit auch die Korrektheit der Implementierung sicherzustellen.

4.6.3 Stresstests

Der Dateisystem-Benchmark *IOzone* [24] wurde benutzt, um viele Anfragen an *wdfs* zu generieren und somit Stresstests unter hoher Belastung durchzuführen. Außerdem wurde es eingesetzt, um verschiedenste Modi für das Öffnen, Lesen und Schreiben von Dateien (wie zufälliges Schreiben oder Rückwärtslesen) zu testen. Die Implementierung bestand den umfangreichsten IOzone-Test, welcher mit dem Befehl `iozone -a -g 2048 -f /tmp/wdfs/iozone-test` gestartet werden kann. Dabei zeigte sich auch, dass bei dem Zugriff auf Subversion mittels *wdfs* der Großteil der Prozessorzeit durch den Apache-Server (bzw. das WebDAV-Subversion-Modul) und nicht durch das Dateisystem verbraucht wird.

4.6.4 Test der Speicherverwaltung

Während der Implementierung des Dateisystems wurde das Programm *Valgrind* [38] benutzt, um die Implementierung der Speicherverwaltung zu validieren. Mit Hilfe von Valgrind ist es unter anderem möglich Speicherlecks zu erkennen, Lese- und Schreibzugriff auf noch nicht allozierten oder bereits freigegebenen Speicher zu erkennen und somit die Qualität des Quelltextes zu verbessern.

Kapitel 5

Einsatzmöglichkeiten

Dieses Kapitel beschreibt einige Einsatzmöglichkeiten des implementierten versionierenden Dateisystems für den transparenten Zugriff auf Subversion. Allen Szenarien gehen von der im Abschnitt 2.1.2 auf Seite 7 vorgestellten Konfiguration (Autoversionierung) aus und davon dass *wdfs* mit dem Parameter `-S` für den Zugriff auf alle Subversion-Revisionen eingebunden wurde.

1. Dateisystemzugriff auf Subversion
   Das Dateisystem wird genutzt um Anwendern den Zugriff auf Daten eines Subversion-Repositorys zu ermöglichen. Die Szenarien reichen von einem einzelnen Anwender, der auf ein Repository auf seinem eigenen Rechner bequem per Dateisystem zugreifen möchte, bis hin zu einem zentralen Repository und einer Vielzahl von Anwendern, wobei die Möglichkeiten des Sperrens von Dateien nützlich ist. Es ist auch denkbar, dass Anwender das Dateisystem nicht selbst einbinden, sondern dies vom Systemverwalter durchgeführt wird und den Anwendern Zugriff auf den Mountpoint des Dateisystems gewährt wird.

2. „Dateisystem mit Gedächtnis“
   Anstatt einzelne Projekte in Repositories zu verwalten ist es möglich, alle Daten eines Anwenders dort zu speichern und diese per *wdfs* einzubinden. Ein solches „Dateisystem mit Gedächtnis“ kann helfen, Datenverluste wie das versehentliche Überschreiben oder Löschen von Daten zu verhindern, da die Daten immer als Teil einer älteren Revision erhalten und zugreifbar bleiben. Sogar der Extremfall eines *wdfs*-Home-Verzeichnisses ist denkbar. Wenn das Repository auf einem entfernten Rechner gespeichert ist, bietet es zusätzlich Schutz vor Datenverlusten durch Hardwareausfall und Angriffe Dritter.

3. Bearbeiten einzelner Dateien eines Repositorys
   Der von Subversion implementierte Copy-Modify-Merge-Prozess besitzt die Einschränkung, dass nur ein Checkout eines kompletten Repositorys und nicht einzelner Dateien oder Verzeichnisse durchgeführt

werden kann.[1] Bei sehr großen Repositorys an denen nur kleine Änderungen durchgeführt werden müssen, führt dies zu unnötigem Netzwerkverkehr und erhöht die benötigte Dauer für die Änderungen der Daten. An Stelle eines Checkout kann das Repository mit *wdfs* eingebunden und die Änderungen durchgeführt werden, ohne alle Daten des Repositorys zum Client übertragen zu müssen.

4. Andere Betriebssysteme und *wdfs*
   Der Dateisystemzugriff auf Subversion mittels *wdfs* kann auch für Anwender anderer Betriebssysteme verfügbar gemacht werden. Dafür ist der Mountpoint von *wdfs* mit einem Netzwerkdateisystem (wie SMB/ CIFS) zu exportieren, so dass die Anwender anderer Betriebssysteme darauf zugreifen können. Der Funktionsumfang von *wdfs* bleibt dabei gleichwertig zu dem der direkten Nutzung unter Linux. Allerdings entsteht zusätzlicher Verwaltungsaufwand, so dass der Zugriff abhängig von der zur Verfügung stehenden Netzwerkbandbreite langsamer wird.

5. Subversion und *wdfs* als Wiki-ähnliches System
   Wiki-Systeme erlauben das direkte Bearbeiten von Websites von jedermann durch die Verwendung von HTML-Formularen und der HTTP-Methode `POST`. Jede Version wird gespeichert und bleibt zugreifbar. Für die Verwaltung dieser Änderungen wird allerdings kein Versionskontrollsystem eingesetzt, obwohl diese die dafür benötigten Funktionen bereits implementieren. Mit Hilfe von *wdfs* kann eine Website ähnlich einem Wiki verwaltet werden. Dafür muss der Mountpoint von einem WebDAV-Server exportiert werden, so dass HTTP-Clients lesend auf die Website und über das virtuelle Verzeichnis auch alle vorherigen Versionen der Website zugreifen kann. WebDAV-Clients können die Dateien der Website auch bearbeiten und so neue Versionen erzeugen. Dies ähnelt dem Direktzugriff von WebDAV-Clients auf WebDAV-Subversion-Repositories mit dem Unterschied, dass die Clients wie von Wiki-Systemen bekannt auf alle Versionen der Website zugreifen können.

[1] `http://subversion.tigris.org/faq.html#single-file-checkout`

Kapitel 6

Zusammenfassung

Die im Rahmen dieser Bachelorarbeit entstandene Implementierung eines Dateisystems für den transparenten Zugriff auf das Versionskontrollsystem Subversion erfüllt die in der Einleitung genannten Anforderungen. Die grundlegenden Dateisystemoperationen werden ebenso unterstützt wie der Zugriff auf alle Revisionen eines Subversion-Repositorys und das Sperren von Dateien zum Schutz vor dem Überschreiben von Daten bei Gruppenarbeit.

Die vorgestellten Einsatzmöglichkeiten zeigen, dass die Verwendung eines Dateisystems für den Zugriff auf Subversion eine deutliche Vereinfachung – besonders im Vergleich zu dem kommandozeilenbasierten Zugriff – darstellt. Der Dateisystemzugriff ermöglicht Anwendern von den Vorteilen der Versionskontrolle zu profitieren, ohne deren Nutzung erlernen zu müssen. Der zusätzliche Verwaltungsaufwand der Speicherung der Daten in einem Versionskontrollsystem kann durch die Vorteile der Versionskontrolle aufgewogen werden.

Die Kombination aus Dateisystem und Versionskontrolle ist für beide Konzepte vorteilhaft. Die Funktionalität von Dateisystemen wird erweitert und die Versionskontrolle findet neue Einsatzmöglichkeiten. Die heutige Leistungsfähigkeit der IT-Infrastruktur bezüglich Speicherkapazität, Rechenleistung und Datenübertragung kann durch das Einbinden von Versionskontrollsystemen als Dateisystem sinnvoll genutzt werden, um neuen Anwendergruppen die Vorteile der Versionskontrolle zu erschließen.

Literaturverzeichnis

[1] *Apache HTTP Server Project,* `http://httpd.apache.org/`

[2] *Apache Software License,* Version 1.1, `http://www.opensource.org/licenses/apachepl.php`

[3] *Apache Tomcat,* `http://jakarta.apache.org/tomcat/`

[4] T. Berners-Lee, R. Fielding, H. Frystyk: *RFC 1945: Hypertext Transfer Protocol – HTTP/1.0,* 1996, `ftp://ftp.rfc-editor.org/in-notes/rfc1945.txt`

[5] T. Berners-Lee, R. Fielding, L. Masinter: *RFC 3986: Uniform Resource Identifier (URI): Generic Syntax,* 2005, `ftp://ftp.rfc-editor.org/in-notes/rfc3986.txt`

[6] D. P. Bovet, M. Cesati: *Understanding the Linux Kernel,* First Edition, O'Reilly 2000

[7] *cadaver: command-line WebDAV client for Unix* `http://www.webdav.org/cadaver/`

[8] G. Clemm, J. Amsden, T. Ellison, C. Kaler, J. Whitehead: *RFC 3253: Versioning Extensions to WebDAV,* 2002, `ftp://ftp.rfc-editor.org/in-notes/rfc3253.txt`

[9] B. Collins-Sussman, B. Fitzpatrick, C. Pilato: *Version Control with Subversion,* `http://svnbook.red-bean.com/`

[10] *Concurrent Versions System,* `http://www.nongnu.org/cvs/`

[11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee: *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1,* 1999, `ftp://ftp.rfc-editor.org/in-notes/rfc2616.txt`

[12] *Filesystem in Userspace,* `http://fuse.sourceforge.net/`

[13] *FUSE SSH Filesystem,* `http://fuse.sourceforge.net/sshfs.html`

[14] *GCC – GNU Compiler Collection,* `http://www.gnu.org/software/gcc/gcc.html`

[15] *GIT - Tree History Storage Tool,* http://git.or.cz/

[16] *GLib Reference Manual,* http://developer.gnome.org/doc/API/2.0/glib/index.html

[17] *GNU Arch,* http://gnuarch.org/

[18] *GNU C Library,* http://www.delorie.com/gnu/docs/glibc/

[19] *GNU General Public License,* http://www.fsf.org/licenses/gpl.html

[20] *GNU Lesser General Public License,* http://www.fsf.org/licensing/licenses/lgpl.html

[21] Y. Goland, E. Whitehead, A. Faizi, S. Carter, D. Jensen: *RFC 2518: HTTP Extensions for Distributed Authoring – WEBDAV,* 1999, ftp://ftp.rfc-editor.org/in-notes/rfc2518.txt

[22] L. Haendel: *The Function Pointer Tutorials,* http://www.newty.de/fpt/index.html

[23] *IBM Rational ClearCase,* http://www-306.ibm.com/software/awdtools/clearcase/

[24] *IOzone Filesystem Benchmark,* http://www.iozone.org/

[25] *Jakarta Slide,* http://jakarta.apache.org/slide/

[26] D. E. Knuth: *The Art of Computer Programming: Volume 3, Sorting and Searching* Second Edition, Addison-Wesley 1998

[27] R. Love: *Linux Kernel Development,* Second Edition, Indianapolis 2005

[28] M. Mitchell, J. Oldham, A. Samuel: *Advanced Linux Programming,* First Edition, Indianapolis 2001

[29] *Neon HTTP and WebDAV client library,* http://www.webdav.org/neon/

[30] S. Oualline: *Practical C Programming,* Third Edition, O'Reilly 1997

[31] F. Prilmeier: *Versionskontrolle von Webseiten am Beispiel CVS,* http://aktuell.de.selfhtml.org/artikel/projekt/cvs/index.htm

[32] K. A. Robbins, S. Robbins: *Unix Systems Programming: Communication, Concurrency and Threads,* Pearson Education 2003

[33] J. Slein, F. Vitali, E. Whitehead, U. C. Irvine, D. Durand: *RFC 2291: Requirements for a Distributed Authoring and Versioning Protocol for the World Wide Web,* 1998, ftp://ftp.rfc-editor.org/in-notes/rfc2291.txt

[34] I. Sommerville: *Software Engineering,* Seventh Edition, Harlow 2004

[35] *Subversion Project,* `http://subversion.tigris.org/`

[36] *Subversion License,*
`http://subversion.tigris.org/project_license.html`

[37] A. S. Tanenbaum: *Moderne Betriebssysteme,* 2., überarbeitete Auflage, München 2003

[38] *Valgrind,* `http://valgrind.org/`

[39] *W3C's Java Server Jigsaw,* `http://www.w3.org/Jigsaw/`

[40] Wikimedia Foundation Inc., Wikipedia Deutschland: *ANSI C,*
`http://de.wikipedia.org/wiki/ANSI_C`

[41] Wikimedia Foundation Inc., Wikipedia Deutschland: *Versionskontrolle,*
`http://de.wikipedia.org/wiki/Versionskontrolle`

[42] Wikimedia Foundation Inc., Wikipedia: *ClearCase,*
`http://en.wikipedia.org/wiki/ClearCase`

[43] Wikimedia Foundation Inc., Wikipedia: *GNU build system,*
`http://en.wikipedia.org/wiki/GNU_build_system`

[44] Wikimedia Foundation Inc., Wikipedia: *Revision control,*
`http://en.wikipedia.org/wiki/Revision_control`

Alle in dieser Arbeit ausgewiesenen URLs wurden am 10. September 2005 abschließend auf ihre Erreichbarkeit überprüft. Für die künftige Erreichbarkeit der verlinkten Inhalte kann keine Garantie übernommen werden. Falls eine URL nicht mehr erreichbar sein sollte, können Kopien der Inhalte per E-Mail an *jens@noedler.de* angefordert werden.

Abkürzungsverzeichnis

ANSI	American National Standards Institute
ASCII	American Standard Code for Information Interchange
CIFS	Common Internet File System
CVS	Concurrent Versions System
DeltaV	Versioning Extensions to WebDAV
FUSE	Filesystem in Userspace
GCC	GNU Compiler Collection
GNU	GNU's Not Unix
GPL	GNU General Public License
HTTP	HyperText Transfer Protocol
IETF	Internet Engineering Task Force
ISO	International Organization for Standardization
LGPL	GNU Lesser General Public License
NFS	Network Filesystem
RFC	Request for Comments
SCM	Source Code Managementsystem
SMB	Server Message Block
SSH	Secure Shell
SSL	Secure Sockets Layer
SVN	Subversion
TCP/IP	Transmission Control Protocol/Internet Protocol

UDF	Universal Disk Format
URI	Uniform Resource Identifier
UTF-8	8-bit Unicode Transformation Format
VCS	Version Control System
VFS	Virtual Filesystem
wdfs	WebDAV Filesystem
WebDAV	Distributed Authoring and Versioning Protocol for the World Wide Web
XML	Extensible Markup Language