Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Sven Gärner

Konzeption und Realisierung eines synchronisierenden Peer-to-Peer Dateisystems

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Sven Gärner

Konzeption und Realisierung eines synchronisierenden Peer-to-Peer Dateisystems

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang Master Of Science Informatik
am Studiendepartment Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften

Betreuender Professor: Prof. Dr. Olaf Zukunft
Zweitgutachter: Prof. Dr. Gunter Klemke

Eingereicht am: 28. November 2008

Sven Gärner

Thema der Arbeit
Konzeption und Realisierung eines synchronisierenden Peer-to-Peer Dateisystems

Stichworte
Synchronisation, automatische Synchronisation, Dateisystem, verteiltes Dateisystem, Versionsverwaltung, Peer-to-Peer Netzwerk, Peer-to-Peer Architektur

Kurzzusammenfassung
Ziel dieser Arbeit ist die Entwicklung eines Konzepts und einer prototypischen Implementierung eines synchronisierenden Peer-to-Peer Dateisystems. Dieses synchronisierende Dateisystem erlaubt die automatische Synchronisation der enthaltenen Dateien und Verzeichnisse mit anderen synchronisierenden Dateisystemen. Alle Dateisysteme, die ihre Dateien und Verzeichnisse untereinander synchronisieren, bilden zusammen ein Peer-to-Peer Netzwerk, in dem alle Teilnehmer gleichberechtigt sind. Zentrale Komponenten sind für die Nutzung nicht notwendig. Die einzelnen Teilnehmer müssen einmalig miteinander bekannt gemacht werden.

Jedes synchronisierende Dateisystem innerhalb des Peer-to-Peer Netzwerks enthält nach erfolgter Synchronisation die gleichen Dateien und Verzeichnisse. Der Anwender kann daher jederzeit, auch wenn keine Verbindung zum Peer-to-Peer Netzwerk besteht, auf alle Dateien und Verzeichnisse zugreifen. Die Synchronisation erfolgt automatisch im Hintergrund, dabei werden auftretende Konflikte erkannt und, falls möglich, gelöst. Solange keine Konflikte auftreten, muss der Anwender nur seine Dateien und Verzeichnisse in dem synchronisierenden Dateisystem verwalten und alle verbundenen synchronisierenden Dateisysteme werden automatisch aktualisiert.

Sven Gärner

Title of the paper

Design and Implementation of a Synchronising Peer-to-Peer Filesystem

Keywords

synchronisation, automatic synchronisation, filesystem, distributed filesystem, version control system, peer-to-peer network, peer-to-peer architecture

Abstract

This paper describes the development of a concept and a prototypical implementation of a synchronising peer-to-peer filesystem. Through this filesystem all containing files and directories automatically synchronise with other synchronising filesystems. Those filesystems form a peer-to-peer network in which all participants share equal rights. Therefore central components are not required. Before synchronising for the first time each participant has to be informed about the existence of the others.

After synchronising, each filesystem within the peer-to-peer network contains the same files and directories. The user can access all files and directories at any time, even if a connection with the peer-to-peer network can not be established. The synchronisation runs in the background, conflicts are detected and solved, if possible. As long as no conflicts are detected, the user just has to use the synchronising filesystem. Other connected synchronising filesystems are automatically updated.

Inhaltsverzeichnis

1. Einleitung 1

2. Grundlagen der Synchronisation 3
   - 2.1. Synchronisation . . . 4
     - 2.1.1. Gleichheit von Verzeichnissen . . . 5
     - 2.1.2. Gleichheit von Dateien . . . 7
   - 2.2. Arten der Synchronisation . . . 14
     - 2.2.1. Unidirektionale Synchronisation . . . 16
     - 2.2.2. Bidirektionale Synchronisation . . . 18
     - 2.2.3. Synchronisation mit Hilfe des Dateisystems . . . 19
   - 2.3. Konflikterkennung . . . 20
   - 2.4. Zusammenfassung . . . 20

3. Möglichkeiten der Synchronisation 22
   - 3.1. Windows Aktenkoffer . . . 22
   - 3.2. rsync . . . 23
   - 3.3. Unison . . . 25
   - 3.4. CVS und Subversion . . . 27
   - 3.5. Mercurial . . . 30
   - 3.6. Coda . . . 33
   - 3.7. Ficus . . . 36
   - 3.8. Ivy . . . 39
   - 3.9. Zusammenfassung . . . 40

4. Szenario 43
   - 4.1. Lösungsansatz . . . 45

5. Konzeption des synchronisierenden Peer-to-Peer Dateisystems 47
   - 5.1. Synchronisation . . . 49
   - 5.2. Konflikterkennung . . . 50
   - 5.3. Konfliktlösung . . . 53
   - 5.4. Dateisystem . . . 56
     - 5.4.1. Synchronisation . . . 59
     - 5.4.2. Konflikterkennung . . . 65
     - 5.4.3. Konfliktlösung . . . 67
     - 5.4.4. Optimierungsmöglichkeiten . . . 68

5.5. Kommunikation 70
5.5.1. Kommunikation des synchronisierenden Dateisystems 71
5.6. Zusammenfassung 73

6. Implementierung 75
6.1. Integration externer Bibliotheken 75
6.2. Komponenten 80
6.2.1. Dateisystem Implementierung 80
6.2.2. Protokollierung 86
6.2.3. Dateizugriff 93
6.2.4. Synchronisation 94
6.3. Aktueller Stand der Entwicklung 99
6.3.1. Optimierungsmöglichkeiten 102
6.4. Testen der Implementierung 103

7. Fazit 106

8. Ausblick 108

A. Anhang 111
A.1. Inhalt der beigelegten CD-ROM 111

Literatur 112

Tabellenverzeichnis

3.1. Eigenschaften des *Windows Aktenkoffers* 23
3.2. Eigenschaften von *rsync* 25
3.3. Eigenschaften von *Unison* 27
3.4. Eigenschaften von *CVS* und *Subversion* 29
3.5. Eigenschaften von *Mercurial* 33
3.6. Eigenschaften von *Coda* 36
3.7. Eigenschaften von *Ficus* 38
3.8. Eigenschaften von *Ivy* 40
3.9. Eigenschaften aller vorgestellten Anwendungen 40

4.1. Eigenschaften der neu zu entwickelnden Anwendung 45

5.1. Auszug einiger protokollierter Operationen 63
5.2. Konflikterkennung 67

6.1. Tabellendefinition der Tabelle *vc_history* 87
6.2. Definition der Tabelle *vc_sync* 95
6.3. Übersicht der implementierten Komponenten des synchronisierenden Peer-to-Peer Dateisystems 102

Abbildungsverzeichnis

3.1. Erzeugung der lokalen *Sandbox* 28
3.2. Übermittlung der lokalen Änderungen von Computer 1 und Abbruch der Übermittlung der Änderungen von Computer 2 28
3.3. Versionshistorie einer Datei in zwei *Repositories* mit unterschiedlicher Entwicklung ab Version 4 31
3.4. Import einer neuen Version 4 eines anderen *Repository* als Version 5 . . 31
3.5. Zusammenführung (*merge*) der Versionen 4 und 5 einer Datei in die neue Version 6 32

5.1. Architektur und Einsatz verschiedener synchronisierender Peer-to-Peer Dateisysteme 47
5.2. Logische Topologie des Beispiels aus Abbildung 5.1 (S. 47) 48
5.3. Verkettung mehrerer synchronisierender Verzeichnisse und Änderung der Datei `letter.tex` bei zwei Teilnehmern 52
5.4. Verzögertes Auftreten des Konflikts erst bei der Synchronisation zwischen *Workstation* und *privatem Notebook* 52
5.5. Synchronisation der neuen Version `letter v3` nach Auflösung des Konflikts 53
5.6. Kreuzweise Synchronisation der Versionen 55
5.7. Schichtenarchitektur und Systemintegration des synchronisierenden Peer-to-Peer Dateisystems 57
5.8. Komponenten des synchronisierenden Peer-to-Peer Dateisystems 58
5.9. Transaktionales Verhalten während der Protokollierung 61
5.10. Beide Zweige der Datei `/test` (siehe Tabelle 5.2 (S. 67)) 67
5.11. Zusammenführung beider Zweige in eine neue Version 68
5.12. Entfernung der Operationen des linken Zweiges 69

6.1. Architektur von *FUSE* und Integration der Dateisystem-Schnittstelle . . 82
6.2. Auszug *fuse* Basisklasse und konkrete, abgeleitete Implementierung *syncfs* 85
6.3. Klassenhierarchie der Operationen 89
6.4. Aufbau des Bytestroms 91

Listings

2.1. Verzeichnis `~/pictures` . . . 15
2.2. Veränderte Dateien in `~/pictures` und `/mnt/pictures` . . . 15
2.3. Veränderte Dateien in `~/pictures` und `/mnt/pictures` . . . 15

5.1. Zwei in Konflikt stehende Versionen einer Datei . . . 55

6.1. Beispiel für den Einsatz des *Scope Guards* . . . 79
6.2. Typsicherheit durch Verwendung von Templates . . . 92
6.3. Beispiel für das asynchrone Empfangen von Daten über eine Netzwerkschnittstelle mit *boost::asio* . . . 96
6.4. Einhängen eines synchronisierenden Dateisystems . . . 99

1. Einleitung

Die Anzahl der Computer pro Anwender nimmt stetig zu (vgl. Satyanarayanan u. a. 1993, S. 3), so dass die Notwendigkeit der Synchronisation ausgewählter und wichtiger Daten zwischen diesen Computern steigt. Entsprechend stellen Pierce und Vouillon fest:

> The gradual shift from many users per computer to many computers per user has been accompanied by a significant increase in replication of important data. Replicating data ensures its availability during periods of disconnected operation, reduces latency during connected operation, and helps protect against loss due to system failures or user errors. (Pierce und Vouillon 2004, S. 1)

Obwohl die beiden vorstehenden Zitate jeweils vier und sogar 15 Jahre alt sind, haben sie dennoch nichts von ihrer Bedeutung und Aktualität verloren. Im Gegenteil: Gerade in Zeiten steigender Verkäufe mobiler Computer, wie zum Beispiel im Bereich der Notebooks und Netbooks, die aufgrund ihrer geringen Maße und fehlender optischer Laufwerke eher als Zusatzgeräte angeschafft werden, sind diese Aussagen aktueller denn je, wie ein kürzlich im *Handelsblatt* erschienener Artikel (vgl. Handelsblatt 2008) belegt.

Ziel dieser Arbeit ist die Entwicklung einer Konzeption und die Umsetzung dieser in Form eines Prototypen einer automatischen Synchronisation.

Zunächst werden die Grundlagen einer Synchronisation von Dateien und Verzeichnissen zwischen verschiedenen Computern oder Datenträgern erläutert (Kapitel 2 (S. 3)), bevor anschließend (Kapitel 3 (S. 22)) verschiedene Anwendungen zur Synchronisation vorgestellt werden. Diese Betrachtung wird mit Sicherheit nicht alle verfügbaren Anwendungen berücksichtigen, dennoch schafft sie einen guten Überblick über verschiedene Ansätze und Umsetzungen, wie eine Synchronisation erfolgen kann. Insbesondere wenn die Synchronisation automatisiert werden soll, reduziert sich die Anzahl der Möglichkeiten. Die Notwendigkeit einer automatischen Synchronisation wird in dieser Arbeit aufgegriffen.

In dieser Arbeit liegt der Fokus nicht generell auf der Synchronisation beliebiger Daten, sondern ausschließlich auf der Synchronisation von Dateien und Verzeichnissen

eines Benutzers. Dazu zählen in erster Linie Dokumente, Bilder einer Digitalkamera oder digitialisierte Musik. Für einzelne Bereiche existieren bereits weit verbreitete Lösungen, wie beispielsweise das im Mobiltelefonen eingesetzte *SyncML* (vgl. Schiller 2003, S. 433 ff.; Open Mobile Alliance 2008).

Verfügbare Anwendungen, die einem Anwender die Synchronisation seiner Dateien zwischen mehreren Computern ermöglichen, müssen von diesem bislang manuell aufgerufen werden, um Synchronisation durchzuführen. Bevor in dieser Arbeit verschiedene Ansätze und entsprechende, bereits existierende Anwendungen zur Synchronisation näher betrachtet werden, soll zunächst jedoch die Synchronisation von Dateien und Verzeichnissen und damit verbundene Probleme untersucht werden.

Nach Vorstellung und Bewertung der verschiedenen Anwendungen, werden zunächst Probleme bei der Nutzung der verschiedenen Anwendungen beleuchtet, bevor anschließend ein Konzept entwickelt wird, um möglichst effizient und automatisch die Dateien – wie Dokumente, Bilder oder digitalisierte Musik – eines Anwenders zu synchronisieren. Dabei werden die Anforderungen an eine automatische Synchronisation näher betrachtet. Schließlich wird anhand der vorgestellten Anforderungen an eine Synchronisation ein Konzept detailliert erarbeitet.

Im Anschluss an das Konzept wird die prototypische Implementierung des erarbeiteten Konzepts vorgestellt. Dabei werden die Abweichungen gegenüber dem Konzept aufgezeigt und begründet.

Es folgte eine Schlussbetrachtung des synchronisierenden Dateisystems und es wird ein Ausblick auf mögliche Verbesserungen, Optimierungen und mögliche Erweiterungen der Implementierung und des Konzepts gegeben.

2. Grundlagen der Synchronisation

In diesem Kapitel wird die Synchronisation von Dateien und Verzeichnissen betrachtet. Zunächst wird definiert, was unter der Synchronisation von Dateien und Verzeichnissen in dieser Arbeit verstanden wird. Anschließend werden verschiedene Arten der Sychronisation von Dateien und Verzeichnissen vorgestellt. Dabei wird ausführlich betrachtet, welche Stellen zu beachten sind und wo Probleme auftreten könnten.

Eine *Synchronisation von Verzeichnissen* erfolgt zwischen mindestens zwei Verzeichnissen. Die meisten Anwendungen verwenden der Übersichtlichkeit halber zwei Verzeichnisse, daher werden in dieser Arbeit auch immer zwei Verzeichnisse betrachtet. Alle in diesen Verzeichnissen enthaltenen Dateien und Unterverzeichnisse werden in die Synchronisation mit eingeschlossen. Die in den zu synchronisierenden Verzeichnissen enthaltenen Dateien und Unterverzeichnisse werden fortan als Objekte bezeichnet. Wenn also zwei Verzeichnisse synchronisiert werden sollen, ist es das eigentliche Ziel, alle enthaltenen Objekte zu synchonisieren. Nach erfolgreicher Ausführung einer Synchronisation zweier Verzeichnisse sind in beiden Verzeichnissen die gleichen Objekte vorhanden und diese sind identisch. Ein Verzeichnis ist somit vergleichbar mit einer Kopie des anderen Verzeichnisses.

Um eine *Synchronisation von Verzeichnissen* durchzuführen, ist es notwendig, die beteiligten Objekte zu identifizieren. Die Identifikation eines Objekts durch den Anwender erfolgt über den Pfad und den Namen des Objekts. Da der Pfad der zu synchronisierenden Verzeichnisse unterschiedlich sein kann, sollte der Pfad des Objekts nur zum zu synchronisierenden Verzeichnis absolut sein. Die entsprechende Definition soll mit Hilfe eines Beispiels verdeutlicht werden.

Definition 1 *Sollen die beiden Verzeichnisse* `/home/user/sync_a` *und* `/mnt/sync_b` *miteinander synchronisiert werden und existiert in beiden jeweils ein Objekt mit dem absoluten Pfad*

- `/home/user/sync_a/path/to/file` *und*
- `/mnt/sync_b/path/to/file`*,*

so lautet der zur Identifikation verwendete Pfad in beiden Fällen

- `/path/to/file`.

Da beide Objekte über den gleichen Namen identifiziert werden, werden diese beiden Objekte zur Synchronisation herangezogen.

2.1. Synchronisation

Der Begriff *Synchronisation von Verzeichnissen* sagt nur aus, dass zwei Verzeichnisse synchronisiert werden sollen. Wie dies geschehen soll, sagt dieser Begriff jedoch nicht aus. Daher soll definiert werden, wie dieser Begriff in dieser Arbeit verstanden wird. Der Einfachheit halber erfolgt die Definition mit Hilfe von zwei Verzeichnissen, die miteinander synchronisiert werden sollen. Diese Definition lässt sich entsprechend auch auf mehr als zwei Verzeichnisse erweitern.

Nachfolgend soll nun definiert werden, welche Bedingungen erfüllt sein müssen, um zwei Verzeichnisse als synchronisiert – also identisch – anzusehen.

Definition 2 *Zwei Verzeichnisse A und B sind dann synchronisiert, wenn alle in beiden Verzeichnissen enthaltenen Objekte synchronisiert sind. Die Objekte werden dazu entsprechend der Definition 1 (S. 3) identifiziert. Dazu muss gelten, dass*

1. *alle in A enthaltenen Objekte auch in B enthalten sind und*
2. *alle in B enthaltenen Objekte auch in A enthalten sind.*

Sei M_A die Menge aller im Verzeichnis A enthaltenen Objekte, M_B entsprechend die Menge aller im Verzeichnis B enthaltenen Objekte und x jedes beliebige Objekt (Datei oder Verzeichnis), dann muss nach der Synchronisation gelten:

$$x \in M_A \land x \in M_B$$
$$\forall x : (x \in M_A \Leftrightarrow x \in M_B)$$

Jedes beliebige Objekt x muss also sowohl in der Menge M_A und als auch in der Menge M_B enthalten sein. Daraus folgt:

$$M_A = M_B$$
$$\Leftrightarrow M_A \subseteq M_B \land M_B \subseteq M_A$$

Beide Verzeichnisse M_A und M_B sind dann identisch.

Diese Definition lässt offen, wie die Gleichheit zweier Objekte definiert ist. Sie definiert nur, welche Bedingungen erfüllt sein müssen, damit zwei Verzeichnisse als synchronisiert angesehen werden können.

Nachfolgend soll diese Definition erweitert werden, indem definiert wird, wann zwei Objekte gleich sind.

2.1.1. Gleichheit von Verzeichnissen

In diesem Abschnitt wir die Gleichheit von Verzeichnissen definiert. Verzeichnisse enthalten einerseits wenige Daten – im Grunde werden nur Informationen über das Verzeichnis gespeichert (Meta-Daten) – und andererseits dienen sie hauptsächlich zur strukturierten Speicherung von Dateien.

Verzeichnisse enthalten im Allgemeinen folgende Meta-Daten:

- Liste der enthaltenen Dateien
- Erzeugungs-, Modifikations- und letzte Zugriffszeit
- Besitzerinformationen (Benutzer und Gruppe)
- Berechtigungen für Besitzer, Gruppen und andere Benutzer

Alle Meta-Daten lassen sich relativ leicht auf Gleichheit überprüfen. Das Heranziehen der verschiedenen Zeitinformationen zur Feststellung der Gleichheit von Verzeichnissen, ist nicht sinnvoll. Hierzu müssten die Uhren der beteiligten Computer, für die normalerweise ein Quarzkristall verwendet wird, synchron laufen, aber es „[...] kann nicht garantiert werden, dass die Kristalle in unterschiedlichen Computern alle mit genau derselben Frequenz oszillieren.“ (Tanenbaum und van Steen 2008, S. 264)

> Ein Problem in verteilten Systemen und Computernetzwerken im Allgemeinen ist, dass es den Begriff einer globalen, gemeinsam genutzten Uhr nicht gibt. (Tanenbaum und van Steen 2008, S. 300)

Eine Synchronisation der Uhren aller beteiligten Computer ist nicht trivial und wird aufwändiger, wenn mobile Computer daran beteiligt sind. Diese sind möglicherweise zu selten oder zu kurz im Netzwerk eingebunden, als dass eine Synchronisation der Uhren erfolgen könnte. Die Verwendung der Zeitinformationen zur Feststellung der Gleichheit von Verzeichnissen ist daher sehr unzuverlässig und soll nicht weiter berücksichtigt werden.

Die Daten über Besitzer und Gruppe, von Verzeichnissen sowie die entsprechenden Berechtigungen lassen sich leicht vergleichen, wenn lokal – etwa mit einem externen Datenträger – synchronisiert wird. Werden jedoch Verzeichnisse zwischen unterschiedlichen Computern oder sogar unterschiedlichen Betriebssystemen synchronisiert, lassen sich die Besitzerinformationen und Berechtigungen nicht mehr ohne weiteres vergleichen. Beispielsweise könnte die Benutzerkennung des Anwenders als numerischer Wert unter dem einen Betriebssystem 1000 sein, während sie unter dem anderen Betriebssystem 1007 ist. Eine Möglichkeit besteht darin, diese Information beim Vergleichen nicht zu berücksichtigen, da ansonsten ein Konflikt entstünde. Eine andere Möglichkeit besteht darin, eine Zuordnung der Benutzerkennungen während der Synchronisation vorzunehmen.

Verwendet der Anwender beispielsweise unterschiedliche Betriebssysteme, könnte sogar der Fall eintreten, dass sich diese Informationen überhaupt nicht sinnvoll vergleichen lassen, da ein System beispielsweise natürliche Zahlen und das andere System *UUIDs (**U**niversally **U**nique **Id**entifier)*[1] (vgl. Microsoft, Refactored Networks, LLC und DataPower Technology, Inc. 2005) verwendet.

Die Berechtigungen sind bei dem Einsatz unterschiedlicher Betriebsysteme ebenfalls nicht immer direkt vergleichbar. Unix-artige Betriebssysteme etwa verwenden ein einfaches Berechtigungssystem. Jedem Verzeichnis (und auch jeder Datei) werden Lese-, Schreib- und Ausführungsrechte für den Besitzer, eine Gruppe und alle anderen Benutzer zugeordnet (vgl. Tanenbaum 2003*b*, S. 804 ff.). *Microsoft Windows*[2] erlaubt stattdessen, mehreren Benutzern und Gruppen detailliertere Berechtigungen zuzuweisen. Außerdem können einzelnen Benutzern und Gruppen Berechtigungen explizit entzogen werden (vgl. Tanenbaum 2003*b*, S. 899 ff.).

Da diese Arbeit nur die Synchronisation der Daten eines Benutzers betrachtet, können unterschiedliche Benutzerkennungen einfach ignoriert und bei der Betrachtung der Gleichheit von Verzeichnissen übergangen werden. Es wird davon ausgegangen, dass die Benutzerkennung des Anwenders auf allen beteiligten Computern identisch ist.

Diese Einschränkung vereinfacht die Definition für die Gleichheit von Verzeichnissen.

Definition 3 *Die Meta-Daten zweier Verzeichnisse a und b sind dann gleich, wenn gilt:*

$$r(a) = r(b)$$

[1] siehe auch http://en.wikipedia.org/wiki/Uuid
[2] Microsoft Website http://www.microsoft.com/

Die Funktion $r(x)$ ermittelt die Berechtigungen für ein beliebiges Verzeichnis x. Weiterhin muss gelten, dass die enthaltenen Dateien gleich sind. Die folgenden Mengen F_a und F_b definieren den entsprechenden Inhalt des jeweiligen Verzeichnisses.

$$F_a = \{\text{alle Dateien des Verzeichnisses } a\}$$
$$F_b = \{\text{alle Dateien des Verzeichnisses } b\}$$
$$F_a = F_b$$

Enthält ein Verzeichnis ein weiteres Verzeichnis, so lässt sich diese Definition entsprechend rekursiv anwenden. Die Gleichheit von Dateien wird hier nicht berücksichtig, da sie später gesondert definiert wird.

Zeit- und Besitzerinformationen bleiben bei dieser Definition unberücksichtigt.

Im nächsten Abschnitt soll die Gleichheit von Dateien betrachtet werden. Da diese nicht nur aus Meta-Daten bestehen, existieren verschiedene Möglichkeiten, die Gleichheit zu ermitteln.

2.1.2. Gleichheit von Dateien

Die Definition für die Gleichheit von Dateien ist komplexer als die vorangegangene Definition für die Gleichheit von Verzeichnissen. Die Meta-Daten von Dateien sind fast mit den Meta-Daten von Verzeichnissen identisch. Dateien enthalten jedoch keine anderen Dateien, so dass diese Information entfällt. Für Dateien werden jedoch ebenso die folgenden Meta-Daten gespeichert:

- Erzeugungs-, Modifikations- und letzte Zugriffszeit
- Besitzerinformationen (Benutzer und Gruppe)
- Berechtigungen für Besitzer, Gruppen und andere Benutzer

Für die Feststellung der Gleichheit dieser Informationen gelten die gleichen Bedingungen, wie dies für Verzeichnisse in Kapitel 2.1.1 (S. 5) beschrieben ist.

Dateien enthalten jedoch neben den Meta-Daten noch einen Inhalt, der für Anwender und Anwendungen hauptsächlich von Bedeutung ist. Den Inhalt zweier Dateien auf Gleichheit zu untersuchen, ist nicht trivial und soll in den nächsten Abschnitten näher untersucht werden.

Syntaktische Gleichheit

Die syntaktische Gleichheit von Dateien wird ausschließlich mit Hilfe der einzelnen Bytes einer Datei festgestellt; das Format der Datei, also ob es sich beispielsweise um ein Bild handelt, wird dabei nicht berücksichtigtq. Um die syntaktische Gleichheit zweier Dateien festzustellen, könnten beispielsweise beide Dateien parallel eingelesen und sequentiell alle Bytes verglichen werden. Dieser Vergleich ist recht einfach umzusetzen, jedoch bei großeren Dateien nicht sehr effizient. Insbesondere wenn eine der Dateien auf einem anderen Computer vorliegt, würde ein entsprechender Vergleich dem Übertragen der kompletten Datei gleichkommen.

Sehr viel effizienter ist es, eine Prüfsumme über die Bytes der beiden Dateien zu berechnen und diese anschließend zu vergleichen. Eine einfache, jedoch ungünstige Methode wäre es, die einzelnen Bytes nacheinander bitweise zu verknüpfen, beispielsweise mit Hilfe der exklusiv-oder Verknüpfung. Dieses Methode ist ungünstig, da sich unterschiedliche Bytefolgen finden lassen, die die gleiche Prüfsumme und somit Kollisionen erzeugen. Die berechnete Prüfsumme muss allerdings eindeutig sein, so dass für zwei unterschiedliche Dateien – genauer unterschiedliche Inhalte – nicht die gleiche Prüfsumme berechnet werden kann. Zur Berechnung einer Prüfsumme sollten kryptografische Hash-Funktionen eingesetzt werden, weil für diese erhöhte Anforderungen gelten, wie Schmeh am Beispiel digitaler Signaturen erklärt:

> Um eine Hashfunktion für digitale Signaturen verwenden zu können, müssen Alice und Bob an diese folgende Zusatzanforderung stellen: Es muss für Angreifer Mallory praktisch unmöglich sein, Kollisionen herbeizuführen. Diese Anforderung ist nicht trivial, weil Kollisionen immer existieren, sofern es mehr Urbilder als Hashwerte gibt. Eine Hashfunktion, die genannte Zusatzanforderung erfüllt, wird als **kryptografische Hashfunktion** bezeichnet.[3] (Schmeh 2007, S. 201)

Auch wenn Schmeh krytpografische Hash-Funktionen für die Erstellung digitaler Signaturen einsetzt, können diese ebenso für den Vergleich zweier Dateien verwendet werden. Der Einsatz ermöglicht eine effiziente Berechnung einer nahezu eindeutigen Prüfsumme beziehungsweise eines Hash-Werts und so die Überprüfung zweier Dateien auf Gleichheit.

Die Darstellung der Anforderung an kryptografische Hash-Funktionen mit Hilfe einer Gleichung von Tanenbaum und van Steen soll nachfolgend eingeführt werden, da später die Definition der Gleichheit von Dateien entsprechend vorgenommen wird.

[3] Im Bereich der Kryptografie kommunizieren häufig Alice und Bob, während Mallory versucht, die Kommunikation der beiden zu manipulieren (vgl. Schmeh 2007, S. 10).

> Schließlich haben kryptografische Hash-Funktionen auch die Eigenschaft der starken Kollisionsresistenz, d.h. für ein gegebenes H ist es nicht möglich, über Berechnungen zwei verschiedene Eingabewerte m und m' zu finden, für die $H(m) = H(m')$ gilt. (Tanenbaum und van Steen 2008, S. 427)

Das *FreeBSD* Ports-System (vgl. FreeBSD Documentation Project 2008, S. 109) verwendet beispielsweise die Hash-Funktionen *SHA1* und *SHA256* (vgl. Schmeh 2007, S. 210 ff.), um sicherzustellen, dass der Download der Software-Quellen den erwarteten Inhalt besitzt. *rsync* (vgl. Tridgell 1999, S. 49) verwendet ebenfalls Hash-Funktionen, um effizient Unterschiede zwischen Dateien zu ermitteln und anschließend beim Kopieren möglichst wenig Datei zu übertragen (Kapitel 3.2 (S. 23)).

Abschließend soll definiert werden, wie die syntaktische Gleichheit von Dateien festgestellt wird. Die Gleichheit der Meta-Daten der Dateien ist entsprechend der Gleichheit für Verzeichnisse definiert, mit der Ausnahme, dass Dateien keine weiteren Objekte enthalten können.

Definition 4 *Die Funktion $r(x)$ ermittelt die Berechtigungen für eine beliebige Datei x. Für die Meta-Daten muss also lediglich gelten:*

$$r(a) = r(b)$$

Zwei Dateien a und b sind dann syntaktisch gleich, wenn gilt:

$$h(a) = h(b)$$

$h(x)$ bezeichnet dabei eine kryptografische Hash-Funktion wie beispielsweise SHA256*, die auf den Inhalt der Datei x angewendet wird.*

Im nächsten Abschnitt soll mit der semantischen Gleichheit eine andere Möglichkeit des Vergleichs untersucht werden.

Semantische Gleichheit

Semantische Gleichheit zwischen Dateien festzustellen, ist nicht so generisch und effizient möglich, wie dies bei der syntaktischen Gleichheit funktioniert, da das Format der jeweiligen Datei erkannt und interpretiert werden muss. Kumar und Satyanarayanan beschreiben beispielsweise die Aktualisierung und Konfliktlösung eines Kalenders mit Hilfe eines generischen Hilfsprogramms, wie es beispielsweise von *CVS* (vgl. GNU CVS; *Open*CVS) oder *Subversion* (vgl. Subversion) eingesetzt wird.

> [...] If the appointments do not overlap, a utility program could merge the two replicas even though a conflict exists at the file granularity. [...] (Kumar und Satyanarayanan 1993, 1. Motivation)

In dem zitierten Beispiel wird semantisch eine Lösung des Konflikts ermöglicht, obwohl syntaktisch ein Konflikt, also keine automatische Lösung, möglich ist.

Daraus lässt sich folgern, dass zwei Dateien semantisch identisch sind, obwohl sie es syntaktisch nicht sind.

Text-Dateien können beispielsweise relativ einfach erzeugt werden, so dass diese semantisch gleich, syntaktisch jedoch unterschiedlich sind. Speichert man einen Text unter einem unix-artigen System, so enthält die resultierende Datei als Zeilenumbruch nur das ASCII-Zeichen[4] 10 ($\backslash n$), unter *Microsoft Windows* erzeugt enthält diese Datei als Zeilenende die ASCII-Zeichen 10 und 13 ($\backslash n \backslash r$) und unter einem *Apple* System wie *MacOS 9* die ASCII-Zeichen 13 und 10 ($\backslash r \backslash n$). Betrachtet man diese Dateien mit einem Editor unter dem jeweiligen System oder vergleicht Ausdrucke der Dateien, so sehen diese identisch aus. Sie sind auch semantisch identisch. Alle Zeilenumbrüche sind an den gleichen Positionen und die Anzahl an Leerzeilen stimmt ebenfalls. Dennoch erzeugen alle Dateien unterschiedliche kryptografische Hash-Werte, da die einzelnen Bytes der Datei unterschiedlich sind. Sie sind somit syntaktisch unterschiedlich.

Um die semantische Gleichheit zweier Dateien festzustellen, müssen diese vorher in eine normalisierte und somit vergleichbare Form überführt werden. Diese normalisierte Form könnte beispielsweise ein *parse tree* (vgl. Sedgewick 1988, S. 305 ff.) sein, der abhängig vom Dateiformat der jeweiligen Datei ist. Sollte die normalisierte Form der beiden Dateien identisch sein, so sind auch die ursprünglichen Dateien identisch. Entsprechend der Definition für die syntaktische Gleichheit (siehe Kapitel 2.1.2 (S. 8)), soll ebenfalls eine Definition für die semantische Gleichheit erfolgen.

Definition 5 *Sei $n(x)$ eine Funktion, die eine normalisierte Form des ihr übergebenen Dateiinhalts einer beliebigen Datei x erzeugt. Zwei Dateien a und b sind dann semantisch gleich, wenn gilt:*

$$n(a) = n(b)$$

[4] ASCII steht für American Standard Code for Information Interchange und stellt eine Kodierung für die Darstellung eines Alphabets dar.

Ließe sich ein kryptographische Hashwert für die normalisierte Form einer Datei berechnen, so würde dann auch gelten:

$$h(n(a)) = h(n(b))$$

Anhand des vorangegangen einfachen Beispiels der Textdatei ist deutlich zu erkennen, dass die Verwendung semantischer Gleichheit als Grundlage für eine Synchronisation von Dateien einen erheblichen Aufwand darstellt. Jedes zu unterstützende Dateiformat muss interpretiert und in eine normalisierte Form überführt werden können, die semantische Gleichheit erkannt und bei der Synchronisation berücksichtigt werden.

CVS (GNU CVS; *Open*CVS) vergleicht beispielsweise zwei Versionen einer Datei zeilenweise, um die Unterschiede zu ermitteln. Allerdings ist dieser Vergleich auf Textdateien beschränkt, wie Vesperman feststellt:

> Die Methode funktioniert allerdings nicht bei Binärdateien, da diese normalerweise keine durch Carriage Return-Zeichen abgegrenzten Textzeilen enthalten. (Vesperman 2004, S. 66)

Dieses Vorgehen ist entspricht einem semantischen Vergleich jedoch nur teilweise, weil das eigentliche Format der Datei unberücksichtigt bleibt. Es ist somit nur ein sehr einfacher semantischer Vergleich. Innerhalb einer Zeile werden alle anderen Bytes direkt verglichen, ohne das Format zu berücksichtigen.q

Aufgrund des erheblichen Aufwands bei der Entwicklung, jedes zu unterstützende Dateiformat interpretieren zu können, soll die semantische Gleichheit in dieser Arbeit nicht weiter berücksichtigt werden. Stattdessen soll nur die syntaktische Gleichheit als Grundlage für die Gleichheit von Dateien verwendet werden.

Gleichheit mit Hilfe von Dateisystem-Operationen

Eine gänzlich andere Art, die Gleichheit von Dateien zu definieren, ist es, die Entwicklung einer Datei zwischen der Erzeugung und der Zerstörung zu betrachten und die ausgeführten Dateisystem-Operationen zum Vergleich heranzuziehen.

Betrachtet man eine Datei aus Sicht eines Dateisystems, so entwickelt sich eine Datei anhand der auf ihr ausgeführten Operationen. Diese Operationen lassen sich wie folgt klassifizieren:

1. Operationen, die Dateiinhalte oder Verzeichnisse verändern
    - `write()`

  - `unlink()`
  - `mkdir()`
  - ...

2. Operationen, die keine Dateiinhalte oder Verzeichnisse verändern
   - `read()`
   - `stat()`
   - `readdir()`
   - ...

3. Operationen, die Dateiinformationen (Meta-Daten) verändern
   - `chmod()`
   - `read()`
   - `write()`
   - ...

Einige Operationen tauchen in mehreren Gruppen auf, wie beispielsweise `read()` oder `write()`. Werden Daten aus der Datei gelesen (`read()`), dann wird in der Regel die Zeit des letzten Zugriffs in den Meta-Daten der Datei aktualisiert[5]. Auf den Dateiinhalt hat dies jedoch keinerlei Auswirkungen.

Würde man die Operationen, die zur aktuellen Version einer Datei geführt haben (beispielsweise `write()`), speichern und erneut ausführen, würde man eine identische Datei erhalten. Aus diesem Grund sollen die vorhergehenden Definition für die Gleichheit einer Datei ergänzt werden:

Definition 6 *Sei O_a die Menge der Operationen, die zur aktuellen Version der Datei a geführt hat und entsprechend sei O_b als die Menge der Operationen der Datei b definiert. Zwei Dateien sind dann gleich, wenn die Menge der Operationen, die zur aktuellen Version einer Datei geführt haben, gleich sind. Dann gilt:*

$$
\begin{aligned}
& O_a = O_b \\
\Leftrightarrow & O_a \subseteq O_b \wedge O_b \subseteq O_a
\end{aligned}
$$

Die hier verwendete Menge muss jedoch besondere Eigenschaften besitzen:

[5] Dateisysteme, die schreibgeschützt oder mit speziellen Optionen eingebunden werden, ändern diese Meta-Daten nicht.

1. *Elemente der Menge dürfen mehrfach vorkommen.*

2. *Die Elemente müssen sich sortieren lassen.*

Diese beiden Eigenschaften können hinzugefügt werden, indem jedes Element eine Identifikation erhält. Diese Eigenschaft ermöglicht eine eindeutige Kennzeichnung für jedes Element, so dass keine zwei Elemente identisch sind. Wird für die Idenfikation beispielsweise eine natürliche Zahl genommen, die für jede neue Operation inkrementiert wird, so kann eine Sortierung der Elemente vorgenommen werden.

Somit wird aus der Menge an Operationen und Menge der natürlichen Zahlen eine Ordnungsrelation.

Die natürliche Zahl kann in diesem Fall auch als logische Uhr nach Lamport angesehen werden (vgl. Tanenbaum und van Steen 2008, S. 275), da sie die von Lamport definierte happens-before *Beziehung darstellt.*

Diese Definition erlaubt es, dass zwei Dateien nach der Definition syntaktischer Gleichheit identisch sind, nach der Definition der Operationen jedoch nicht. Dies lässt sich sehr leicht nachvollziehen, wenn man beispielsweise eine Textdatei erzeugt und mehrfach modifiziert. Dann ergibt sich eine Menge von Dateisystem-Operationen. Erzeugt man durch Kopieren dieser Datei eine neue Datei, dann sind beide Dateien syntaktisch und semantisch identisch. Die Menge der Dateisystem-Operationen sind jedoch nicht identisch, da die Menge der Dateisystem-Operationen der neu erzeugten Datei in den meisten Fällen aus den folgenden Operationen bestehen wird:

1. `open()`

2. `write()` ($1, \ldots, n$ Aufrufe, abhängig von der Dateigröße)

3. `close()`

Daher stellt diese Definition der Gleichheit von Dateien eine Möglichkeit dar, die jedoch syntaktisch und sogar semantisch gleiche Dateien nicht als identisch erkennt. Für den Anwender ist die Definition der syntaktischen Gleichheit die entscheidende Definition, da sich der Anwender ausschließlich für den Inhalt der Datei und möglicherweise noch für die Meta-Daten interessiert und diese für einen Vergleich heranzieht. Die Menge der Dateisystem-Operationen wird er nie für einen Vergleich verwenden, vor allem, weil diese Menge nirgends hinterlegt ist.

Des Weiteren gibt es noch Operationen, die kommutativ ausgeführt werden können, dass heißt die Reihenfolge der Ausführung der Operationen ist unerheblich. Die Ausführung der Operationen produziert bei den entsprechenden Fällen immer das gleiche Resultat, führt also zur syntaktisch identischen Datei.

Führt man beispielsweise eine `write()` Operation aus – man schreibt also Daten in eine Datei – und benennt diese anschließend um (`rename()`), so erhält man eine identische Datei, wenn man die Datei zuerst umbennent und anschließend die Änderungen in die Datei schreibt. Entscheidend ist nur, dass die Namensänderung entsprechend bei der `write()` Operation berücksichtigt wird, die Datei also trotz Umbenennung eindeutig identifizierbar bleibt.

Zwei `write()` Aufrufe sind allerdings nicht kommutativ, da sich die Reihenfolge, in der die Operationen ausgeführt werden, direkt auf den sich ergebenden Inhalt der Datei auswirken. Eine unterschiedliche Reihenfolge der Ausführung könnte syntaktisch verschiedene Dateien produzieren.

Nachdem ausführlich dargestellt wurde, wie die Gleichheit von Verzeichnissen und Dateien festgestellt werden kann, können diese Grundlagen nun dazu verwendet werden, ihren Einsatz bei der Synchronisation von Verzeichnissen im nächsten Kapitel näher zu betrachten.

2.2. Arten der Synchronisation

In diesem Abschnitt soll vorgestellt werden, wie eine Synchronisation zweier Verzeichnisse erfolgen könnte. Dabei wird, aufbauend auf die bisher entwickelten Grundlagen bezüglich der Gleichheit von Verzeichnissen und Dateien, entwickelt, welche Informationen notwendig sind, um eine Synchronisation zuverlässig und ohne Datenverlust durchzuführen. Mögliche Probleme und der Umgang mit diesen sollen ebenfalls untersucht werden, um die Anforderungen an eine möglichst automatische Synchronisation zu ermitteln.

In Listing 2.1 (S. 15) ist ein Verzeichnis inklusive der enthaltenen Objekte dargestellt. Dieses Verzeichnis wird nachfolgend zur Verdeutlichung bei der Synchronisation und zur Behandlung von möglichen Konflikten herangezogen. Das Verzeichnis `~/pictures` soll immer mit dem Verzeichnis `/mnt/pictures` synchronisiert werden, so dass nach erfolgter Synchronisation beide Verzeichnisse identisch sind. Die in Verzeichnis `/mnt/pictures` enthalten Objekte werden in den folgenden Beispielen jeweils explizit dargestellt.

```
~/pictures/
  wallpaper/
  icons/
    gnu-emacs.png
    xterm.png
    psi.png
```

Listing 2.1: Verzeichnis `~/pictures`

Die folgenden Beispiele stellen verschiedene Ausgangssituationen vor einer Synchronisation dar. Diese Ausgangssituationen werden später verwendet, um die Anforderungen an eine Synchronisation und mögliche Konflikte während einer Synchronisation zu verdeutlichen.

Beispiel 1 *Das Verzeichnis* `~/pictures` *enthält die in Listing 2.1 (S. 15) dargestellten Objekte und das Verzeichnis* `/mnt/pictures` *ist leer.*

Beispiel 2 *In diesem Fall sind beide Verzeichnisse bis auf wenige Dateien identisch. Dieser Zustand entsteht, wenn zuerst eine Synchronisation erfolgt ist und anschließend in beiden Verzeichnissen, einzelne Objekte verändert wurden. Dieser Zustand erfordert ein Aktualisieren der Dateien von* `~/pictures` *nach* `/mnt/pictures` *und in die andere Richtung. Die*

```
~/pictures                      /mnt/pictures
   icons/gnu-emacs.png
   icons/psi.png
                                   psi.png
```

Listing 2.2: Veränderte Dateien in `~/pictures` und `/mnt/pictures`

Mengen der veränderten Objekte beider Verzeichnisse sind in diesem Fall disjunkt.

Beispiel 3 *Dieser Fall hat als vorhergehenden Zustand ebenfalls zwei identische Verzeichnisse, jedoch wurden in beiden Verzeichnisse die gleichen Objekte verändert.*

```
~/pictures                      /mnt/pictures
   icons/gnu-emacs.png             icons/gnu-emacs.png
   icons/psi.png                   icons/psi.png
```

Listing 2.3: Veränderte Dateien in `~/pictures` und `/mnt/pictures`

Es existieren verschiedene Klassen der Synchronisation, die unidirektionale und die bidirektionale Synchronisation, die nachfolgend vorgestellt werden. Dabei werden auftretende Probleme und mögliche Lösungswege aufgezeigt.

2.2.1. Unidirektionale Synchronisation

Die unidirektionale Synchronisation bezeichnet die Synchronisation in eine Richtung. Es werden die Objekte eines Quellverzeichnisses verwendet, um ein Zielverzeichnis auf den Stand des Quellverzeichnisses zu aktualisieren. Da die Synchronisation nur in eine Richtung erfolgt, werden nur Änderungen an den Dateien und Verzeichnissen des Quellverzeichnisses berücksichtigt. Eventuell vorhandene und auch geänderte Dateien des Zielverzeichnisses werden überschrieben.

Nachfolgend soll anhand der zuvor in Abschnitt 2.2 (S. 14) definierten Beispiele das Verhalten einer unidirektionalen Synchronisation verdeutlicht werden.

Im ersten Fall (Beispiel 1 (S. 15)), in dem das Zielverzeichnis `/mnt/pictures` leer ist, gestaltet sich die Synchronisation sehr einfach, da alle Objekte des Quellverzeichnisses (`~/pictures`) rekursiv in das Zielverzeichnis (`/mnt/pictures`) kopiert werden können. Konflikte treten daher in diesem Fall nicht auf.

Der zweite und dritte Fall (Beispiel 2 (S. 15) und Beispiel 3 (S. 15)) lassen sich mit der unidirektionalen Synchronisation nicht ohne Datenverlust synchronisieren.

Für den zweiten Fall (Beispiel 2 (S. 15)), wenn die Mengen der veränderten Objekte disjunkt sind, ließe sich eine Synchronisation ermöglichen, wenn beispielsweise nur „neuere“ Objekte der Quellseite auf die Zielseite kopiert werden. „Neuer“ würde bedeuten, dass die Modifikationszeiten der Objekte verglichen und die auf Quellseite aktuelleren Objekte kopiert werden müssten. Dies funktioniert gut, solange die Synchronisation lokal, beispielsweise auf eine externe Festplatte, erfolgt. Sobald jedoch zwischen mehreren Computern synchronisiert wird, ist dieser Vergleich der Modifikationszeiten nicht unbedingt als zuverlässig anzusehen.

Geht beispielsweise die Uhr des Computers auf der Quellseite der Synchronisation im Vergleich mit der Uhr des Computers auf der Zielseite nach, würden keine Objekte auf der Zielseite aktualisiert werden. Alle Objekte der Zielseite würden als „neuer“ angesehen werden, auch wenn dies nicht der Fall wäre. Um die Zuverlässigkeit dieses Vergleichs zu erhöhen, ist eine Synchronisation der Uhren der an der Synchronisation beteiligten Computer notwendig, da die Uhren zweier Computer nie exakt synchron laufen (vgl. Tanenbaum und van Steen 2008, S. 264). Um die Uhren verschiedener Computer zu synchronisieren, existieren verschiedene Algorithmen, die jedoch nicht trivial

sind und regelmässig ausgeführt werden müssen. Das *Network Time Protocol (NTP)* (vgl. Tanenbaum und van Steen 2008, S. 270 ff.) implementiert einen entsprechenden Algorithmus, ist weit verbreitet und ermöglicht, die Uhren verschiedener Computer in einem Netzwerk zu synchronisieren. Gerade wenn mobile Computer zum Einsatz kommen und diese nur kurzzeitig im gleichen Netzwerk erreichbar sind, ist eine zuverlässige Uhrensynchronisation kaum möglich. Dennoch kann dieser kurze Zeitraum nicht dazu verwendet werden, einfach die Uhrzeit neu zu stellen, da beispielsweise die Uhrzeit nicht rückwärts laufen darf (vgl. Tanenbaum und van Steen 2008, S. 269 ff.).

Es ist daher nur möglich, die unidirektionale Synchronisation einzusetzen, wenn sichergestellt ist, dass sich die zu synchronisierenden Objekte nur auf einer der beiden beteiligten Seiten geändert haben.

Um nun nicht bei jeder Synchronisation alle Objekte der Quellseite auf die Zielseite kopieren zu müssen, sind verschiedene Optimierungen denkbar. Ein Vergleich der Modifikationszeiten wäre möglich, soll aber aufgrund der schon erwähnten Problematik der Uhrensynchronisation der beteiligten Computer hier nicht weiter berücksichtigt werden.

Eine andere Möglichkeit der Optimierung besteht darin, dass kryptografische Hash-Werte der Datei auf der Quell- und auf der Zielseite berechnet werden, diese verglichen und nur die Objekte kopiert werden, deren Hash-Werte unterschiedlich sind. Dadurch hat auch eine nachgehende Uhr eines Computers keinen Einfluss auf die Synchronisation. Diese Optimierung erfordert zwar immer noch, dass Objekte als Ganzes kopiert werden müssen, allerdings müssen nur veränderte Objekte kopiert werden. Eine darauf aufbauende Optimierung verwendet der von Tridgell entwickelte und in *rsync* implementierte Algorithmus, der in Kapitel 3.2 (S. 23) vorgestellt wird.

Werden auf der Quellseite neue Objekte angelegt, so werden diese bei zukünftigen Synchronisationen ebenso berücksichtigt wie schon vorhandene Objekte. Da keinerlei Zustände zwischen zwei Synchronisationsläufen gespeichert werden, könnte auch jede Datei als neue Datei angesehen werden. Werden auf der Quellseite Objekte entfernt, so bleiben sie auf der Zielseite erhalten. Um entfernte Objekte auf der Zielseite ebenfalls zu entfernen, müssen auf der Quell- und auf der Zielseite die Mengen der zu synchronisierenden Objekte gebildet werden, um dann mit Hilfe der Bildung der Differenzmenge die Objekte zu identifizieren, die auf der Zielseite entfernt werden können.

Eine Synchronisation zwischen zwei verschiedenen Computern ist mit der unidirektionalen Synchronisation möglich, jedoch nicht unbedingt flexibel zu verwenden. Es muss immer darauf geachtet werden, dass nur jeweils auf einer der zu synchroni-

sierenden Seiten Objekte verändert werden. Am besten lässt sich eine unirektionale Synchronisation verwenden, wenn ausschließlich eine Seite der Synchronisation verändert wird, wie beispielsweise im Falle einer einfachen Datensicherung.

2.2.2. Bidirektionale Synchronisation

Dieser Abschnitt beschäftigt sich mit der bidirektionalen Synchronisation. Da die bidirektionale Synchronisation – wie der Name schon sagt – in zwei Richtungen synchronisiert, kann man nicht mehr von einer Quell- und einer Zielseite sprechen. Vielmehr werden die Objekte zweier Verzeichnisse synchronisiert, so dass nach Ausführung der Synchronisation die beiden Verzeichnisinhalte identisch sind (vgl. Definition 2 (S. 4)).

Um erfolgreich zwei Verzeichnisse synchronisieren zu können, reichen die Informationen über die syntaktische Gleichheit von Dateien (vgl. Definition 4 (S. 9)) nicht aus, da die Anwendung niemals eindeutig feststellen könnte, in welchem Verzeichnis sich das aktuelle Objekt befindet und somit auch unklar wäre, in welchem Verzeichnis das entsprechende Objekt zu aktualisieren wäre. Bei der unidirektionalen Synchronisation ist dies unproblematisch, da durch Angabe der Quell- und Zielseite implizit eine Richtung der Synchronisation vorgegeben wird. Dies ist hier jedoch nicht der Fall.

Um dennoch möglichst automatisch synchronisieren zu können, müssen Zustandsinformationen zwischen den Ausführungen der Synchronisation gespeichert werden. Werden beispielsweise kryptografische Hash-Werte verwendet, um zu ermitteln, ob zwei Objekte unterschiedlich sind, lässt sich ohne die Speicherung von Zustandsinformationen nur feststellen, ob die Objekte identisch sind oder nicht. Diese Information reicht jedoch nicht aus, um zu entscheiden, welches der beiden Objekte zu aktualisieren ist.

Wird für jedes Objekt der kryptografische Hash-Wert als Zustandsinformation abgelegt, so lässt sich durch einen Vergleich dieses Wertes mit dem momentanen kryptografischen Hash-Wert der Datei feststellen, ob dieses Objekt verändert wurde. Werden diese Zustandsinformationen für jedes Verzeichnis, das an der Synchronisation beteiligt ist, abgelegt, so lässt sich ebenfalls feststellen, welche Objekte zu aktualisieren sind, welche Objekte einen Konflikt verursachen und welche Objekte nicht verändert wurden.

Existieren Objekte in einem Verzeichnis, für die keine Zustandsinformationen vorhanden sind, so sind diese Objekte neu und sollten fortan in die Synchronisation mit eingeschlossen werden. Ebenso können fehlende Objekte, für die noch Zustandsinformationen existieren, als entfernt angesehen und in dem anderen zu synchronisierenden Verzeichnis entfernt werden.

Nur in dem Fall, dass noch keine Zustandsinformationen existieren, muss der Anwender gefragt werden, wie zu synchronisieren ist.

Mit Hilfe der beiden Verzeichnisse `~/pictures` und `/mnt/pictures` aus Kapitel 2.2 (S. 14) (siehe auch Listing 2.1 (S. 15)) soll diese Art der Synchronisation genauer betrachtet werden.

Im ersten Fall (siehe Beispiel 1 (S. 15)) muss der Anwender – sofern keine Zustandsinformationen vorhanden sind – gefragt werden, wie die Synchronisation erfolgen soll. Sind Zustandsinformationen vorhanden, so wird entsprechend dieser Daten synchronisiert.

Im zweiten Fall (siehe Beispiel 2 (S. 15)) kann ebenfalls automatisch synchronisiert werden, wenn Zustandsinformationen vorhanden sind. Gibt es keine Zustandsinformationen, muss der Anwender entscheiden, welche Objekte aktualisiert werden sollen.

Im letzten Fall (siehe Beispiel 3 (S. 15)) kann nicht automatisch synchronisiert werden, da gleiche Objekte in beiden Verzeichnissen verändert wurden, so dass der Anwender die Änderungen zusammenführen muss, bevor synchronisiert wird oder entscheiden muss, welches Objekt überschrieben werden soll.

2.2.3. Synchronisation mit Hilfe des Dateisystems

Eine weitere Möglichkeit ist, die Objekte nicht anhand ihres Inhalts, sondern anhand der ausgeführten Operationen zu betrachten. Dabei werden alle verändernden Operationen aufgezeichnet und diese zur Synchronisation verwendet, indem die aufgezeichneten Operationen erneut für die zu synchronisierenden Objekte ausgeführt werden.

Ein großer Vorteil, der sich aus der Aufzeichnung der Operationen ergibt, besteht darin, dass bekannt ist, welche Objekte verändert wurden. Daher ist es nicht mehr notwendig, vor der Synchronisation das gesamte Verzeichnis nach veränderten Objekten zu durchsuchen.

Nachteilig ist, dass im Vergleich zu einer Anwendung, die das zu synchronisierende Verzeichnis auf veränderte Objekte durchsucht, wesentlich mehr Informationen zu speichern sind. Zudem stellt sich die Frage, wie lange die Operationen eines Objekts gespeichert werden sollen. Würden sie nie gelöscht werden, würde sich ein versioniertes Dateisystem ergeben, das es ermöglicht, jede Version eines Objekt wieder zu erzeugen. Werden alle Operationen gespeichert, wird einerseits Speicherplatz dafür benötigt und andererseits werden so möglicherweise mehr Informationen abgelegt, als für eine Synchronisation notwendig sind.

2.3. Konflikterkennung

In den bisher genannten Szenarien wurden Konflikte nur pro Objekt erkannt. Wurden beispielsweise zwei namensgleiche Objekte in den beiden zu synchronisierenden Verzeichnissen verändert, so resultierten diese Änderungen in einem Konflikt, der vom Anwender zu beheben war.

Abstrahiert man dieses Vorgehen, so könnten Konflikte auch anhand von Bedingungen erkannt werden. Dies soll an einem Beispiel verdeutlicht werden.

Beispiel 4 *Es könnte folgende Bedingung definiert sein, die erfüllt sein muss, damit kein Konflikt erkannt wird und eine Synchronisation erfolgen kann.*

$$
\begin{aligned}
C &= \{\text{alle veränderten Dateien}\} \\
x &= \text{'icons/gnu-emacs.png'} \\
y &= \text{'icons/psi.png'} \\
\nexists(x,y) &: (x \in C \land y \in C)
\end{aligned}
$$

Es darf entweder nur das Objekt `icons/gnu-emacs.png` *oder* `icons/psi.png` *verändert werden, jedoch dürfen nie beide Objekte verändert werden.*

Um eine entsprechende Konflikterkennung zur Verfügung zu stellen, muss der Anwender jedoch eine Möglichkeit haben, die Bedingungen für eine erfolgreiche Synchronisation zu definieren. Dies könnte beispielsweise eine formale Sprache sein. Die Auflösung eines Konflikts wird zudem komplexer, weil mehrere Objekte daran beteiligt sein könnten. Nicht zu vergessen ist, dass die Definition der Bedingungen auch widersprüchlich sein kann, was von der Implementierung abgefangen werden muss. Schließlich muss auch beachtet werden, dass die definierten Bedingungen ebenfalls zu synchronisieren sind, wodurch wiederum Konflikte – diesmal allerdings pro Objekt – auftreten können.

2.4. Zusammenfassung

Zusammenfassend lässt sich festhalten, dass die unidirektionale Synchronisation für viele Einsatzbereiche (zum Beispiel zur Datensicherung) gut geeignet ist. Ein Vorteil ist, dass keine weiteren Zustandsinformationen gepflegt werden müssen, so dass entsprechende Anwendungen unkompliziert und ohne Konfiguration einsetzbar sind. Da die Synchronisation direkt nach dem Start der Anwendung erfolgen kann, muss der Anwender vorsichtig beim Einsatz sein, um einem Datenverlust vorzubeugen.

Die bidirektionale Synchronisation ist sicherer in der Anwendung als die unidirektionale Synchronisation, da Konflikte erkannt werden und somit Datenverluste verhindert werden können. Nachteilig ist, dass Zustandsinformationen für jedes Objekt gepflegt werden müssen. Insbesondere bei der Synchronisation vieler Objekte erhöht sich die Ausführungszeit, da vor der eigentlichen Synchronisation die Zustandsinformationen mit den zu synchronisierenden Verzeichnissen verglichen werden müssen, um zu entscheiden, welche Objekte zu synchronisieren sind.

Die Synchronisation mit Hilfe der Dateisystem Operationen stellt einen anderen Ansatz dar, bei dem die Dateien nicht während der Synchronisation auf Änderungen überprüft werden müssen, wie bei der bidirektionalen Synchronisation, sondern alle relevanten Änderungen während der Nutzung des Dateisystems protokolliert werden können. Somit können die protokollierten Operationen direkt verwendet werden, um die Dateien zu synchronisieren. Vorteilhaft ist, die direkte Speicherung der für eine Synchronisation relevanten Informationen, so dass diese Informationen nicht erst während der Synchronisation ermittelt werden müssen. Nachteilig ist, dass eine Implementierung sehr nah am Betriebssystem erfolgen muss und somit ist eine Portierung auf unterschiedliche Betriebssysteme nicht so einfach vorzunehmen, wie dies bei einem Anwendungsprogramm der Fall wäre.

3. Möglichkeiten der Synchronisation

In diesem Kapitel sollen verschiedene Anwendungen und Ansätze vorgestellt werden, die eine Synchronisation von mindestens zwei Verzeichnissen ermöglichen. Des Weiteren sollen die verschiedenen Anwendungen und Ansätze bezüglich der Arten der Synchronisation (Kapitel 2.2 (S. 14)) klassifiziert werden. Zunächst soll der *Windows Aktenkoffer* betrachtet werden, da er auf vielen Computern verfügbar ist. Anschließend werden zwei Anwendungen (*rsync* und *Unison*) betrachtet, deren Ziel es ist, möglichst effizient Dateien zu synchronisieren. Bevor abschließend verschiedene Dateisysteme (*Coda*, *Ficsu* und *Ivy*) betrachtet werden, sollen unterschiedliche Anwendungen zur Versionsverwaltung (*CVS*, *Subversion* und *Mercurial*) untersucht werden, die indirekt synchronisieren und alle Änderungen protokollieren.

3.1. Windows Aktenkoffer

Der *Windows Aktenkoffer* ist Bestandteil von *Microsoft Windows* und auch nur unter diesem System nutzbar. Er soll helfen, Dateien zu synchronisieren. Der Name dieser Anwendung spiegelt die verwendete Metapher wider. Der Anwender fügt Dateien, die mobil bearbeitet werden sollen, also unabhängig vom jeweiligen Computer, dem *Aktenkoffer* hinzu. Die Originaldateien müssen allerdings auf dem Computer erhalten bleiben.

Der *Aktenkoffer* wird aus Sicht des Anwenders wie ein Verzeichnis verwendet. Zunächst werden die zu synchronisierenden Dateien in den *Aktenkoffer* kopiert, anschließend können die Dateien innerhalb des Aktenkoffers bearbeitet werden. Danach wird dieser beispielsweise auf ein Notebook kopiert. Um nun die Änderungen zu synchronisieren, muss der *Aktenkoffer* erneut auf den Computer kopiert werden, der die Originaldateien enthält. Dort kann nun der *Aktenkoffer* geöffnet und die Dateien synchronisiert werden. Welche Dateien verändert wurden, wird anhand der Modifikationszeit der Dateien erkannt, so dass diese Erkennung problematisch sein kann.

Es wird dabei erkannt, wenn die Originaldateien und nicht die Dateien innerhalb des *Aktenkoffers* verändert wurden, so dass diese Änderungen ebenfalls synchronisiert werden.

Wurden allerdings die Originaldatei und die entsprechende Kopie im *Aktenkoffer* verändert, so dass ein Konflikt auftritt, dann wird dieser erkannt, kann jedoch nicht mit Hilfe des *Aktenkoffers* behoben werden. Um den Konflikt zu lösen, muss der Anwender die Änderungen zusammenführen und eine der beiden manuell Versionen überschreiben. Es besteht aber nicht Möglichkeit, während der Synchronisation zu entscheiden, dass eine der Dateien überschrieben werden soll.

Beim *Windows Aktenkoffer* erfolgt die Synchronisation immer zwischen den Originaldateien und dem *Aktenkoffer*, der aber keinem festen Ort zugeordnet ist. Vielmehr kann der *Aktenkoffer* auch verschoben werden, ohne dass die Synchronisationsmöglichkeiten verloren gehen.

Um die Dateien zu synchronisieren, muss zuerst der *Aktenkoffer* geöffnet und anschließend die Synchronisation vom Anwender gestartet werden. Es können mehrere *Aktenkoffer* verwendet werden, die Synchronisation erfolgt jedoch immer nur mit den Originaldateien, die nicht verschoben werden dürfen. Andernfalls fehlen den entsprechenden Dateien im *Aktenkoffer* die Originaldateien und die Dateien im *Aktenkoffer* gelten als verwaist.

Zusammenfassend lassen sich die folgenden Eigenschaften festhalten:

Synchronisationsrichtung	bidirektional
Konflikterkennung	✓
Konfliktlösung	✗
automatische Synchronisation	✗
Server erforderlich	✗

Tabelle 3.1.: Eigenschaften des *Windows Aktenkoffers*

3.2. rsync

rsync (vgl. Tridgell 1999; rsync) wurde von Tridgell im Rahmen seiner Doktorarbeit entwickelt und kann am besten mit einem Zitat, das auf der *rsync* Website zu finden ist, beschrieben werden:

> rsync is an open source utility that provides fast incremental file transfer. (rsync, Website)

Der *rsync* Algorithmus wurde entwickelt, weil der Autor viel Zeit damit verbrachte, darauf zu warten, seinen Quelltext über eine Modemleitung zu kopieren (vgl. Tridgell 1999, S. 49). Es waren jedoch auf beiden Seiten der Modemleitung Teile des Quelltextes vorhanden, so dass in der Regel nur Änderungen hätten übertragen werden müssen. Diese Idee implementiert der *rsync*-Algorithmus.

rsync kopiert die Änderungen von einer Quell- zu einer Zielseite, arbeitet also unidirektional (siehe Kapitel 2.2.1 (S. 16)). Zuerst wird die zu übertragene Datei auf der Zielseite gesucht. Ist sie dort nicht vorhanden, kann der *rsync* Algorithmus nicht angewandt werden. Die Datei muss komplett übertragen werden. Ist die Datei dort vorhanden, wird sie in nicht überlappende Blöcke definierter und gleicher Größe aufgeteilt (mit Ausnahme des letzten Blocks). Für jeden dieser Blöcke wird eine schwache und eine starke Prüfsumme berechnet (siehe Kapitel 2.1.2 (S. 8)). Diese Informationen werden anschließend zur Quellseite übertragen.

Dort wird versucht, diese Blöcke in der lokalen Datei wiederzufinden. Dabei kommt eine besondere Eigenschaft der schwachen Prüfsumme zum Tragen, wodurch eine effizientere Implementierung ermöglicht wird. Diese besondere Eigenschaft besteht darin, dass sich aus der Prüfsumme der Bytes $X_1, \ldots, X_n$ und den Werten der Bytes X_1 und X_{n+1} die Prüfsumme für die Bytes $X_2, \ldots, X_{n+1}$ berechnen lässt. Diese Eigenschaft wird von Tridgell *rolling property* (vgl. Tridgell 1999, S. 54) oder *rolling checksum* (vgl. Tridgell 1998) genannt. Dadurch ist es möglich, effizient, und zwar in einem Durchlauf, eine Datei nach Blöcken mit bekannter Prüfsumme zu durchsuchen.

Die schwache Prüfsumme hat nicht die Eigenschaften eines kryptografischen Hash-Wertes und ist somit nicht kollisionsresistent. Ist ein Block gefunden worden, wird die starke Prüfsumme für diesen Block berechnet, um sicherzustellen, dass auf beiden Seiten der Synchronisation der gleiche Block referenziert wird. Nur wenn auch diese starke Prüfsumme auf beiden Seiten identisch ist, wird eindeutig der gleiche Block referenziert.

Während der Ermittlung der gleichen beziehungsweise unterschiedlichen Blöcke ist bereits bekannt, welche Bytes übertragen werden müssen. Diese Informationen können dann zusammengestellt werden. Ist die gesamte Datei untersucht worden, können die Informationen über unterschiedliche Bytes zur Zielseite gesandt werden. Auf der Zielseite kann aus der dort vorhandenen Datei und den Informationen über die unterschiedlichen Bytes, die Datei der Quellseite erzeugt werden. Es werden also nur die Unterschiede übermittelt.

Bei einem Einsatz von *rsync* ist entscheidend, dass *rsync* sowohl auf der Quell- als auch auf der Zielseite vorhanden ist. Wenn also Dateien von einem Computer zu einem anderen transferiert werden sollen, muss auf beiden Computer *rsync* installiert sein. Die Vorteile des *rsync*-Algorithmus werden allerdings erst deutlich, wenn Änderungen zu übertragen sind.

Zusammenfassend lassen sich die folgenden Eigenschaften festhalten:

Synchronisationsrichtung	unidirektional
Konflikterkennung	✗
Konfliktlösung	✗
automatische Synchronisation	✗
Server erforderlich	✗

Tabelle 3.2.: Eigenschaften von *rsync*

3.3. Unison

Unison (vgl. Pierce und Vouillon 2004; Unison) synchronisiert im Gegensatz zu *rsync* Verzeichnisse bidirektional. *Unison* hat zudem die besondere besondere Eigenschaft, dass das Verhalten mathematisch spezifiziert wurde:

> An unusual feature of Unison's history is that the engineering of the system has proceeded in parallel with a serious effort to specify its behavior mathematically. (Pierce und Vouillon 2004, S. 1)

Da *Unison* bidirektional synchronisiert, ist es notwendig, Informationen über vorhergehende Synchronisationen zu speichern. Diese Informationen (im *Unison* Sprachgebrauch „Archiv" genannt) sind auf dem Computer abgelegt, auf dem das jeweilige Verzeichnis existiert. Dies setzt voraus, dass auf jedem Computer, der an der Synchronisation beteiligt ist, *Unison* installiert ist. Hieraus ergibt sich der Vorteil, dass das zu synchronisierende Verzeichnis lokal mit dem Archiv verglichen werden kann. Während der Synchronisation müssen dann nur die Informationen dieses Vergleichs ausgetauscht werden, so dass *Unison* auch eingesetzt werden kann, wenn die Netzwerkverbindung wenig Bandbreite zur Verfügung stellt.

Um die zu synchronisierenden Objekte zu ermitteln, verwendet *Unison* Detektoren (*update detectors*) (vgl. Balasubramaniam und Pierce 1998, Kapitel 3), die das Prädikat *dirty()*, also eine Eigenschaft dieses Objekts, berechnen oder ermitteln. Für jedes Objekt

o, für das *dirty(o)* „*true*" liefert, muss überprüft werden, ob der Inhalt der beiden Objekte unterschiedlich ist. Dieses Modul wird bei *Unison reconciler* (abgleichen) genannt. Die einfachste Variante des Prädikats *dirty(o)* liefert „*true*" für jedes Objekt *o*. Jedes Objekt muss also vom *reconciler* überprüft werden (vgl. Balasubramaniam und Pierce 1998, Kapitel 3.2.1)).

Diese Architektur ermöglicht es, das Prädikat für unterschiedliche Betriebssysteme unterschiedlich zu implementieren. Unter unix-artigen Systemen kann der Vergleich die *I-Node* berücksichtigen, um etwa zu erkennen, ob eine Datei umbenannt wurde und eine andere Datei ersetzt (vgl. Balasubramaniam und Pierce 1998, Kapitel 3.2.4) hat. In diesem Fall bleibt die *I-Node* der umbenannten Datei gleich und die überschriebene Datei wird entfernt.

Bei allen veränderten Objekten *o*, bei denen *dirty(o)* „*true*" liefert, muss vom *reconciler* der Inhalt überprüft werden. Diese Objekte werden dann in die Synchronisation eingeschlossen, die vom Anwender zu starten ist, da dieser nicht automatisch auflösbare Konflikte vorher lösen muss.

In diesem Zusammenhang wird als Prädikat auch eine spezielle Implementierung des Prädikats *dirty(o)* erwähnt, die unter *Microsoft Windows* möglich wäre: ein *Online Update Detector* (vgl. Balasubramaniam und Pierce 1998, Kapitel 3.2.5). Dieser Detektor könnte für alle zu synchronisierenden Objekte die Dateisystem Operationen protokollieren. Dieser Detektor wurde von Balasubramaniam und Pierce nur angesprochen, über einen Einsatz in *Unison* wurde jedoch nichts erwähnt. Dennoch könnte diese Variante eine Synchronisation möglicherweise beschleunigen, da das zu synchronisierende Verzeichnis nicht vor jeder Synchronisation rekursiv durchsucht und mit den gespeicherten Informationen verglichen werden müsste.

Zur Übertragung der Objekte verwendet *Unison* ein Verfahren, das dem *rsync*-Algorithmus ähnlich ist. Dieses Verfahren basiert auf der Doktorarbeit von Tridgell, so dass veränderte Objekte effizient übertragen werden.

Aufgrund der bidirektional Synchronisation müssen Zustandsinformationen zwischen verschiedenen Synchronisationen erhalten bleiben. Damit die bidirektional Synchronisation alle Konflikte erkennen kann, müssen zuerst die gespeicherten Zustandsinformationen mit dem Zustand des Verzeichnisses verglichen werden. Anschließend kann synchronisiert werden, und im Falle eines Konflikts kann der Anwender eine Entscheidung treffen, wie *Unison* diesen Konflikt lösen soll.

Wie bei *rsync* auch, muss auf jedem Computer, mit dem Verzeichnisse synchronisiert werden sollen, *Unison* installiert sein.

Die Arbeitsweise von *Unison* ist komplett spezifiziert und *Unison* stellt die Referenzimplementierung dar.

Zusammenfassend lassen sich die folgenden Eigenschaften festhalten:

Synchronisationsrichtung	bidirektional
Konflikterkennung	✓
Konfliktlösung	✓
automatische Synchronisation	✗
Server erforderlich	✗

Tabelle 3.3.: Eigenschaften von *Unison*

3.4. CVS und Subversion

GNU CVS (Concurrent Versioning System) (GNU CVS), die neue Implementierung *OpenCVS* (*Open*CVS) und *Subversion* (Subversion) sollen in diesem Abschnitt gemeinsam betrachtet werden, da sie sehr ähnlich arbeiten (vgl. Collins-Sussman, Fitzpatrick und Pilato 2007, S. XIII). *OpenCVS* stellt eine neue Implementierung von *CVS* dar, das in den letzten Jahren kaum noch gewartet und weiterentwickelt wurde (vgl. *Open*CVS). Daher schließt fortan der Begriff *CVS OpenCVS* mit ein und beide werden als ein System betrachtet.

Beide Systeme sind Versionskontrollsysteme und verwenden eine Client-Server Architektur. Es gibt einige wesentliche Unterschiede zwischen beiden Systemen, die aber im Rahmen dieser Betrachtung nicht entscheidend sind, da sie für den Benutzer kaum sichtbar sind. Beispielsweise werden die Versionsinformationen bei *CVS* pro Datei inkrementiert, während *Subversion* die Versionsinformation pro übermittelter Änderung inkrementiert. Weiterhin verwalten *CVS* und *Subversion* ihre *Repositories*, die zentrale Datenbank mit der Versionshistorie, unterschiedlich. Die Verwendung der Systeme ist jedoch im Großen und Ganzen identisch.

Sollen Dateien zwischen mehreren Computern synchronisiert werden, so muss bei beiden Systemen ein Computer die Rolle des Servers übernehmen. Der Server verwaltet zentral in einem *Repository* alle Dateien und Verzeichnisse inklusive ihrer kompletten Versionshistorie (vgl. Collins-Sussman, Fitzpatrick und Pilato 2007, S. 10; Vesperman 2004, S. 33). Der Anwender arbeitet jedoch nicht direkt mit diesem *Repository*, sondern erzeugt sich lokal eine Kopie einer bestimmten Version (*checkout*), die sogenannte *Sandbox* (vgl. Collins-Sussman, Fitzpatrick und Pilato 2007, S. 14; Vesperman 2004,

S. 33). Abbildung 3.1 (S. 28) verdeutlicht dieses Vorgehen. Zunächst muss die lokale *Sandbox* auf beiden Computer mit einer Kopie der Daten des *Repositories* erzeugt werden.

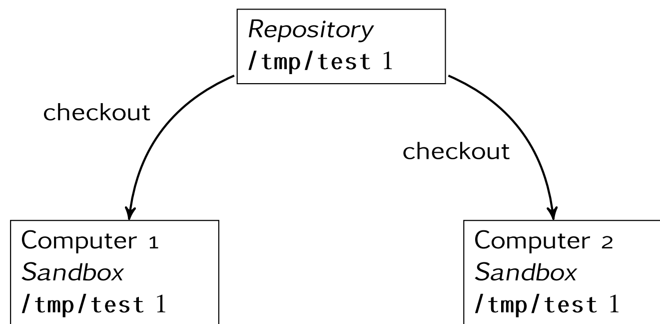

Abbildung 3.1.: Erzeugung der lokalen *Sandbox*

Nach Änderungen an der lokalen Kopie kann Computer 1 seine Änderungen an das *Repository* übermitteln. Computer 2 kann seine Änderungen nicht übermitteln, da diese Version nicht mehr mit der Version des *Repository* übereinstimmt. Daher muss Computer 2 zuerst die Version des *Repository* mit der lokalen Version zusammenführen (vgl. Collins-Sussman, Fitzpatrick und Pilato 2007, S. 15). Abbildung 3.2 (S. 28) verdeutlicht dieses Vorgehen.

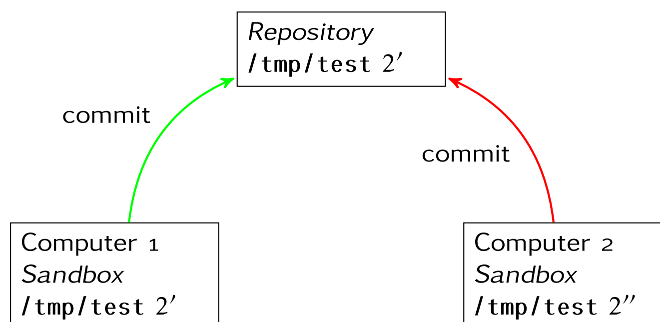

Abbildung 3.2.: Übermittlung der lokalen Änderungen von Computer 1 und Abbruch der Übermittlung der Änderungen von Computer 2

Die gesamte Kommunikation zwischen den *Sandboxes* und somit auch zwischen verschiedenen Computern erfolgt immer über das *Repository*. Sollen Änderungen an das *Repository* übermittelt werden, wird zuerst geprüft, ob nicht neuere Versionen der geänderten Dateien verfügbar sind. Ist dies der Fall, müssen diese zuerst aktualisiert werden. Dabei versuchen sowohl *CVS* als auch *Subversion*, die lokale Version einer Datei

mit der Version des *Repository* zusammenzuführen. Dies funktioniert gut bei Textdateien (vgl. Vesperman 2004, S. 66), sofern sich die Änderungen nicht überschneiden, also gleiche Zeilen in beiden Versionen der Datei verändert wurden. In so einem Fall wird ein Konflikt erkannt, der vom Anwender zu lösen ist. Ein Konflikt tritt dabei immer nur in einer *Sandbox* auf, niemals im *Repository*. *Subversion* ermöglicht es sogar, mit Hilfe einer Statusabfrage vor einer Aktualisierung der *Sandbox* zu überprüfen, ob Konflikte auftreten werden (vgl. Collins-Sussman, Fitzpatrick und Pilato 2007, S. 34 ff.).

Die Synchronisation von Dateien zwischen verschiedenen Computern mit Hilfe von *CVS* oder *Subversion* ist ein eher ungewöhnlicher Anwendungsfall, da nur ein Anwender mit dem System arbeitet. Außerdem wurden die Systeme in erster Linie entwickelt, um die Möglichkeit zu schaffen, vielen Software-Entwicklern den Zugriff auf ein gemeinsames Projekt zu gewähren und dieses weiterzuentwickeln (vgl. Vesperman 2004, S. 3).

Die lokale *Sandbox* des Anwenders ist von allen anderen *Sandboxes* isoliert, so dass auftretende Konflikte auch nur lokal in einer *Sandbox* auftreten und dort behoben werden müssen.

Bei *CVS* und *Subversion* bleiben alle jemals eingestellten Änderungen erhalten. Dadurch besteht die Möglichkeit, jede Änderung – falls nötig – rückgängig zu machen. Da beide Anwendungen für die Verwaltung von Quelltext gedacht sind, funktioniert die Verwaltung einfacher Textdateien am besten. Bei binären Dateien – wie beispielsweise Bildern oder OpenOffice.org Dokumenten – können keine für den Anwender sinnvollen Unterschiede oder Konflikte dargestellt werden, das weder *CVS* noch *Subversion* das jeweilige Format kennen.

Zusammenfassend lassen sich die folgenden Eigenschaften festhalten.

Synchronisationsrichtung	bidirektional
Konflikterkennung	✓
Konfliktlösung	✓
automatische Synchronisation	✗
Server erforderlich	✓

Tabelle 3.4.: Eigenschaften von *CVS* und *Subversion*

3.5. Mercurial

Mercurial (vgl. O'Sullivan 2007; Mercurial) ist – ebenso wie *CVS* und *Subversion* – ein Versionskontrollsystem. Allerdings verwendet *Mercurial* eine Peer-to-Peer Architektur (vgl. Schill und Springer 2007, S. 31 ff.; Tanenbaum und van Steen 2008, S. 63 ff.), so dass jede lokale *Sandbox* gleichzeitig ein komplettes *Repository* enthält.

Zunächst erstellt der Anwender entweder ein leeres *Repository* oder er erzeugt sich eine Kopie eines vorhandenen *Repository*. Das Erstellen einer Kopie eines *Repository* ist vergleichbar mit dem Erzeugen der *Sandbox* bei *CVS* und *Subversion*. Bei *Mercurial* enthält ein Verzeichnis sowohl das *Repository* als auch die *Sandbox*. Die Versionshistorie und Verwaltungsdaten – das *Repository* – werden von *Mercurial* in einem versteckten Unterverzeichnis abgelegt. Dies ist jedoch für den Anwender nicht weiter entscheidend, da die dort abgelegten Daten nicht direkt verändert werden, sondern nur indirekt durch die Nutzung von *Mercurial*.

In diesem Verzeichnis – der *Sandbox* – kann der Anwender nun seine Änderungen vornehmen und in das *Repository* einstellen. Die Funktionsweise ist identisch mit der von *CVS* und *Subversion*, mit dem wesentlichen Unterschied, dass sowohl die *Sandbox* als auch das *Repository* lokal existieren. Daher müssen Änderungen gesondert in andere *Repositories* übertragen werden, sofern dies dem Anwender erlaubt ist. Um die Änderungen zwischen zwei *Repositories* zusammenzuführen, existieren zwei verschiedene Möglichkeiten.

1. Alle lokalen Änderungen werden an das andere *Repository* gesendet (*push*).

2. Alle Änderungen eines anderen *Repositories* werden in das lokale *Repository* importiert und lokal integriert (*pull*).

Setzt ein Anwender *Mercurical* ein, um seine Daten zu synchronisieren, wird er vermutlich mit dem jeweils lokalen *Repository* arbeiten. Da es somit mehrere *Repositories* gibt, die dem Anwender gehören und die er nutzt, bleiben dem Anwender beide zuvor genannten Möglichkeiten der Integration. Die Integration wird mit dem Begriff *merge (zusammenführen)* bezeichnet, da zwei Entwicklungszweige zusammengeführt werden.

Grundsätzlich hängt die verwendete Möglichkeit der Integration der Änderungen davon ab, welche Berechtigungen ein Anwender auf ein entferntes *Repository* hat. Wenn er beispielsweise keine Änderungen in das entfernte *Repository* integrieren kann, dann bleibt nur die Möglichkeit, Änderungen des entfernten *Repositories* in das eigene zu integrieren.

Sind die Änderungen in das *Repository* integriert worden, muss noch die *Sandbox* aktualisiert werden, um mit den neuen Dateien arbeiten zu können.

Da *Mercurial* kein zentrales *Repository* besitzt, besteht die Möglichkeit, dass sich eine Datei in zwei *Repositories* unterschiedlich weiterentwickelt. Dies ist in Abbildung 3.3 (S. 31) dargestellt. Die beiden Graphen stellen jeweils die Entwicklung einer Datei in unterschiedlichen *Repositories* dar. Bis einschließlich Version 3 sind beide Dateien identisch, danach haben sie sich unterschiedlich weiterentwickelt. Version 4 in beiden *Repositories* ist unterschiedlich. In Abbildung 3.4 (S. 31) wurde die Version in das linke *Repository* importiert, anschließend erfolgt die Zusammenführung im linken *Repository*. Diese resultiert in einer neuen Version der Datei (Abbildung 3.5 (S. 32)).

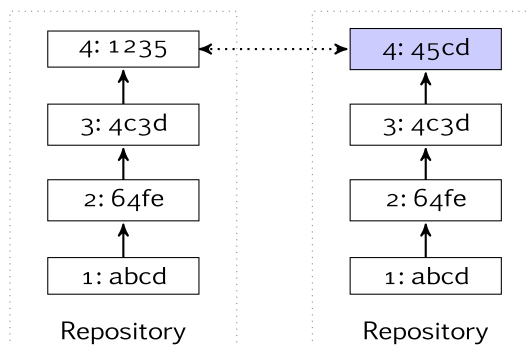

Abbildung 3.3.: Versionshistorie einer Datei in zwei *Repositories* mit unterschiedlicher Entwicklung ab Version 4

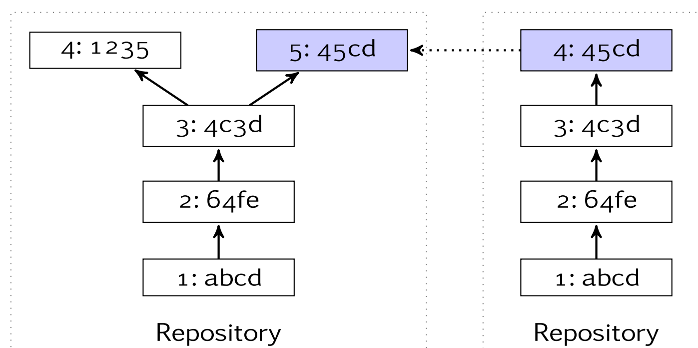

Abbildung 3.4.: Import einer neuen Version 4 eines anderen *Repository* als Version 5

Die Grafik zeigt ebenfalls deutlich, dass es notwendig ist, zusätzlich zu der Versionsnummer eine weitere Identifikation zu verwenden, um unterschiedliche und identische Versionen erkennen zu können.

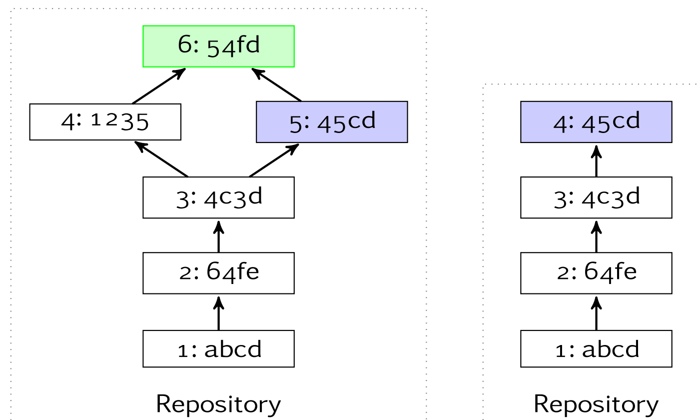

Abbildung 3.5.: Zusammenführung (*merge*) der Versionen 4 und 5 einer Datei in die neue Version 6

Eine Synchronisation erfolgt bei *Mercurial* bidirektional. Der Einsatz eines „zentralen“ *Repositories*, in das alle Änderungen intergriert werden, ist bei mehr als zwei lokalen *Repositories* sinnvoll, da dies die Synchronisation deutlich vereinfacht. Dieses Vorgehen ähnelt dem *Primary Backup Protocol* (vgl. Tanenbaum und van Steen 2008, S. 339 ff.), bei dem ein System alle Schreibvorgänge ausführt und diese dann an die Backup-Systeme propagiert. In diesem Fall würden die Änderungen jedoch nicht propagiert werden, sondern jedes Backup-System (*Repository*), müsste die Änderungen selbstständig synchronisieren. Es ist dann aber nicht mehr notwendig, jedes *Repository* mit jedem anderen zu synchronisieren. Vielmehr existiert ein zentraler Synchronisationspunkt, der von allen anderen *Repositories* genutzt wird.

Auch wenn dies der von *CVS* und *Subversion* verwendeten Client-Server Architektur sehr ähnlich ist, so hat *Mercurial* einen entscheidenden Vorteil gegenüber diesen Versionskontrollsystemen: Es lassen sich jederzeit lokal neue Versionen erzeugen und in das *Repository* einstellen.

Bei *Mercurial* wird lokal immer ein komplettes *Repository* vorgehalten, so dass die *Sandbox* autark genutzt werden kann. Selbst wenn die anderen *Sandboxes* beziehungsweise *Repositories* gelöscht werden, kann lokal problemlos weitergearbeitet werden.

Bei Bedarf können lokal gemachte Änderungen an andere *Repositories* übermittelt werden. Wird dann die entsprechende *Sandbox* aktualisiert, können Konflikte auftreten, die jedoch nur innerhalb dieser *Sandbox* sichtbar und lösbar sind. Andere *Sandboxes* beziehungsweise *Repositories* sind davon nicht betroffen.

Zusammenfassend lassen sich die folgenden Eigenschaften festhalten:

Synchronisationsrichtung	bidirektional
Konflikterkennung	✓
Konfliktlösung	✓
automatische Synchronisation	✗
Server erforderlich	✗

Tabelle 3.5.: Eigenschaften von *Mercurial*

3.6. Coda

Coda (vgl. Coda) wurde an der *Carnegie Mellon University (CMU)* entwickelt. Es ist im Gegensatz zu *rsync* und *Unison* keine Anwendung zur Synchronisiation von Dateien, sondern ein verteiltes Dateisystem. *Coda* besitzt einige wesentliche Eigenschaften, die es von anderen verteilten Dateisystemen, wie beispielsweise den Dateidiensten des *Server Message Block Protocols (SMB, „Windows Dateifreigabe")* (vgl. Server Message Block) und dem *Network Filesystem (NFS)* (vgl. Network File System), unterscheidet.

Coda ist Ende der 1980er aus einer Erweiterung für das *Andrew File Systems (AFS)* (vgl. AFS) hervorgegangen. Da *AFS* an der *CMU* für mehrere tausend Clients eingesetzt wurde (vgl. Tanenbaum und van Steen 2003, S. 678) und es häufig zu Netzwerk- und Serverausfällen kam, sollte die zu entwickelnde Erweiterung ein Weiterarbeiten trotz dieser Probleme ermöglichen (vgl. Braam 1998). Während etwa zweijähriger Erfahrung mit dem Einsatz von *Coda* sind Defizite und neue Anwendungsfälle zu Tage getreten. Insbesondere mobile Computer wurden von Satyanarayanan u. a. als mögliche, neue Umgebung für *Coda* gesehen.

> Portable computers are commonplace today. In conjunction with high- and low-bandwidth cordless networking technology, such computers will soon provide a pervasive hardware base for mobile computing. A key requirement of this new world of computing will be the ability to access critical data regardless of location. (Satyanarayanan u. a. 1993, Kapitel 1)

Der größte Unterschied zu den anderen, vorher genannten verteilten Dateisystemen ist, dass *Coda* es ermöglicht, auf Dateien des Dateiservers zuzugreifen, auch wenn der Client nicht mit dem Server verbunden ist.

Damit dies möglich ist, muss der Client einen Cache verwenden, der die ohne Verbindung zum Server zugreifbaren Dateien enthält. Dieser Cache muss entsprechend dimensioniert sein. Es existieren zwei unterschiedliche Möglichkeiten einen solchen Cache zu füllen (vgl. Schiller 2003, S. 378).

Beim normalen Arbeiten mit *Coda* wird dieser Cache immer als LRU-Cache (Least-Recently-Used)[1] (vgl. Tanenbaum und Woodhull 1997, S. 334 ff.; Tanenbaum 2003*b*, S. 239 ff.) verwendet. Alle Dateien, auf die der Anwender zugreift, werden zuerst komplett in den Cache geladen. Der Server merkt sich, welcher Client welche Datei in den Cache kopiert hat, das sogenannte *Rückrufversprechen (Callback Promise)* (vgl. Tanenbaum und van Steen 2008, S. 563). Anschließend arbeitet der Anwender nur mit dieser lokal vorhanden Version der Datei. Hat der Anwender eine im Cache befindliche Datei verändert, wird der Server – sofern dieser erreichbar ist – über diese Änderung informiert. Sobald der Server diese Information erhalten hat, werden alle Clients, für die ein *Rückrufversprechen* vorliegt, informiert, dass ihre Version veraltet ist, der sogenannte *Rückrufbruch (Callback Break)* (vgl. Tanenbaum und van Steen 2008, S. 563). Der Client weiß nun, dass seine Version der Datei veraltet ist und kann den Cache aktualisieren. Solange also das *Rückrufversprechen* nicht gebrochen wird, ist der Cache des Clients aktuell und der Zugriff kann auf die Datei im Cache erfolgen.

Für einen verbindungslosen Betrieb des Clients, können Dateien und Verzeichnisse als *sticky* (klebrig) markiert werden (vgl. Braam 1998). Entsprechend markierte Dateien und Verzeichnisse werden fortan nicht mehr aus dem Cache verdrängt, so dass ein Arbeiten ohne Verbindung zum Server möglich ist. Damit der Anwender nicht jede Datei, die im verbindungslosen Betrieb verfügbar sein soll, zuerst öffnen muss, damit diese im Cache abgelegt wird, kann das Befüllen des Caches automatisiert werden. Dazu kann der Anwender mittels des *Hoardings* (horten, hamstern), den Cache mit allen als *sticky* markierten Dateien befüllen lassen (vgl. Braam 1998).

Ein weiterer Vorteil, der sich aus der Strategie ergibt, jede Datei zuerst in den lokale Cache zu transferieren, ist, dass ein Arbeiten über instabile Verbindungen wie etwa Funknetzwerke ermöglicht wird. Andere verteilte Dateisysteme sind in solchen Umgebungen fehleranfälliger, so dass entsprechend instabile Verbindung auch zu defekten Dateien führen können, da diese Dateisysteme wie das verbreitete *NFS* (vgl. Network File System) nicht darauf vorbereitet sind. Weiterhin entlastet der Cache den Server, da viele Zugriffe zuerst lokal erfolgen.

Aus Sicht des Anwenders wird bei *Coda* – im Gegensatz zu *NFS* etwa – nicht direkt auf die Server zugegriffen. Alle in diesem Abschnitt genannten Dateisysteme verwenden zwar eine Client-Server-Architektur, aber bei *NFS* oder *SMB* muss der Client wissen, welcher Server ein gewünschtes Verzeichnis exportiert. Dies ist bei *Coda* nicht notwendig. Mit Hilfe des *Domain Name Systems DNS* (vgl. Tanenbaum und van Steen

[1] siehe auch Wikipedia: `http://en.wikipedia.org/wiki/Cache_algorithms`

2008, S. 238 ff.) lassen sich die Namen der *Coda* Server vom Client erfragen. Diese können dann kontaktiert werden, wenn der Client eine Datei öffnen möchte. Der Client sendet eine Anfrage an einen Server und als Antwort erhält er eine eindeutige Kennung für diese Datei. Mit dieser Information kann er den genauen Ort der Datei (Volume und Server) ermitteln und die Datei öffnen. Für den Anwender erfolgt der Zugriff auf die Datei immer über den gleichen Pfad, etwa `/coda/domain.local/user/test.txt`, unabhängig davon, auf welchem Volume und Server die Datei tatsächlich abgelegt wurde.

Ein Volume ist eine spezielle, einem Verzeichnis ähnliche Einheit, die von *Coda* in spezieller Weise verwendet wird. Zunächst dient es dazu, Anwendern spezielle Berechtigungen zuzuordnen, die etwas umfangreicher als die üblichen Berechtigungen bei unix-artigen Systemen sind. Dadurch wird es möglich, Administrationsrechte einzelnen Anwendern zuzuordnen. Weiterhin können Volumes auf verschiedene Server repliziert werden, so dass durch Redundanz eine erhöhte Ausfallsicherheit erreicht werden kann. Wenn ein Volume auf verschiedene Server repliziert wird, kann es möglich sein, dass die Verbindung zwischen den Servern zeitweise unterbrochen wird. Dies beeinträchtigt jedoch nicht den Zugriff der Clients. Wenn die Verbindung zwischen den Servern wieder funktionsfähig ist, können eventuelle Änderungen an den Volumes repliziert werden. Damit dies funktioniert, wird für jede Datei ein Versionsvektor verwendet (vgl. Tanenbaum und van Steen 2008, S. 566), damit die Server erkennen, ob während der Unterbrechung der Kommunikation Änderungen an einzelnen Dateien vorgenommen wurden. Diese Änderungen können dann repliziert werden, sofern kein Konflikt aufgetreten ist, also eine Datei in mehreren Replikas des entsprechenden Volumes verändert wurde. Dieser Konflikt muss dann vom Anwender behoben werden.

Weiterhin kann auch dann ein Konflikt auftreten, wenn beispielsweise zwei Anwender eine Datei ohne Zugriff auf den Dateiserver verändert haben. Der Konflikt tritt jedoch nur auf einem Client auf, da der Datei-Server die Zugriffe nacheinander abarbeitet.

Ein Konflikt kann vom Anwender manuell gelöst werden, indem er dem Server mitteilt, welche Version aktuell ist. Zudem besteht eine Möglichkeit, dies mittels *Application Specific Resolution (ASR)* (vgl. Kumar und Satyanarayanan 1993) zu automatisieren. Dafür können Regeln definiert werden, wie ein Konflikt zu lösen ist.

Da *Coda* eine eigene Benutzerverwaltung nutzt und diese unabhängig von der Benutzerverwaltung des Betriebssystems arbeitet, ist eine Integration von *Coda* aufwändig, da zwei Benutzerverwaltungen zu pflegen sind. Ein daraus resultierendes Problem

ist, dass die Benutzerverzeichnisse der Anwender nicht mit Hilfe von *Coda* verwaltet werden können. Zuerst muss der Anweder sich am System anmelden, um das separate Anmeldeprogramm für die *Coda* Server nutzen zu können. Da die Anmeldung am System aber auch auf das Benutzerverzeichnis zugreift, auf das noch kein Zugriff möglich ist, kann die Anmeldung nicht fehlerfrei erfolgen. Da ein Zugriff auch ohne Verbindung zum Dateiserver möglich ist, andernfalls würde der verbindungslose Betrieb nicht funktionieren, lässt sich dieses Problem in den Griff bekommen. Dazu muss sich der Anwender zunächst ohne Serververbindung anmelden, so dass er auf sein Benutzerverzeichnis zugreifen kann. Danach kann die Anmeldung an den *Coda* Servern erfolgen. Dennoch ist dies keine wirklich praktikable Möglichkeit, da der Anwender jedesmal darauf achten muss, sich ohne Verbindung zum *Coda* Server anzumelden.

Der Betrieb eines dedizierten Servers erhöht deutlich die Anforderungen und den Aufwand, *Coda* zu Synchronisationszwecken einzusetzen. Eine automatische Synchronisation ist jedoch auch kein Ziel, das die *Coda* Entwickler verfolgen, sondern eher eine nützliche Eigenschaft. Daher kann *Coda* nicht so einfach und schnell für diese Zwecke getestet werden, wie beispielsweise *Unison* (siehe Kapitel 3.3 (S. 25)) oder *Subversion* (siehe Kapitel 3.4 (S. 27)).

Wenn *Coda* jedoch eingesetzt wird, dann hat der Anwender sehr wenig Arbeit mit der Synchronisation, da – sofern keine Konflikte auftreten – das synchronisieren der Dateien automatisch erfolgt. Und wenn überwiegend ein Anwender auf die jeweiligen Dateien zugreift, treten in der Regel auch keine Konflikte auf.

Zusammenfassend lassen sich die folgenden Eigenschaften festhalten.

Synchronisationsrichtung	bidirektional
Konflikterkennung	✓
Konfliktlösung	✓(automatisierbar)
automatische Synchronisation	✓
Server erforderlich	✓(dediziert)

Tabelle 3.6.: Eigenschaften von *Coda*

3.7. Ficus

Ficus (Popek und Reiher 1996) ist ein replizierendes Dateisystem und wurde zwischen 1991 und 1996 an der *University of California*, *Los Angeles (UCLA)*, entwickelt. Die

Computer, auf denen die Dateien des Dateisystems repliziert werden, sind in einer Peer-to-Peer-Architektur organisiert (vgl. Schiller 2003, S. 380 f.).

Ficus wurde anscheinend nicht weiterentwickelt und das Projekt im Jahre 1996 beendet. Zudem hat *Sun*[2] sein Betriebssystem *SunOS*[3] inzwischen durch *Solaris* ersetzt. Möglicherweise wären die Anpassungen zu umfangreich gewesen. Der Algorithmus, der die Dateien synchronisiert, wurde zwar in der Anwendung *Rumor* (vgl. Rumor) implementiert, jedoch existiert davon auch nur eine Version aus dem Jahre 2000, die offensichtlich nicht weiterentwickelt wird und auf einem aktuellen Linux System nicht getestet werden konnte.

Da *Ficus* als Dateisystem sehr stark in das Betriebssystem, in diesem Fall *SunOS*, integriert wurde, war es notwendig, zunächst eine Schichtenarchitektur zu entwickeln, um die für eine Synchronisation notwendigen Komponenten einfacher integrieren zu können. Dazu wurde eine Dateisystem-Schnittstelle entwickelt, die der vorhandenen Schnittstelle sehr ähnlich war, jedoch die Möglichkeit bot, einfacher um zusätzliche Funktionalität erweitert werden zu können (vgl. Heidemann und Popek 1991, S. 4 ff.). Diese wurde dann für die Implementierung verwendet. Die neue Schichtenarchitektur ermöglichte die Verwendung des vorhandenen *Unix File System (UFS)*, so dass die Ablage von Dateien und Verzeichnissen an dieses delegiert werden konnte (vgl. Guy u. a. 1990, S. 3 ff.).

Darüber hinaus war es durch die neue Schichtenarchitektur möglich, eine Dateireplikation einzubauen. *Ficus* verwendet eine optimistische Replikationsstrategie, so dass ein Arbeiten mit dem Dateisystem auch möglich ist, wenn nicht alle Computer miteinander verbunden sind (vgl. Heidemann u. a. 1992).

Durch Verwendung der optimistischen Replikationsstrategie können Konflikte auftreten, da mehrere Anwender gleichzeitig die gleiche Datei verändern können. Um diese Konflikte zu erkennen, verwendet *Ficus* einen Versionsvektor (vgl. Guy u. a. 1990, S. 6) und im Falle eines Konflikts wird der Besitzer der Datei informiert (vgl. Guy u. a. 1990). Während einer aktiven Testphase traten dennoch relativ wenig Konflikte auf, da hauptsächlich die Benutzerdaten der einzelnen Anwender repliziert wurden (vgl. Heidemann u. a. 1992): Diese werden fast ausschließlich nur vom Besitzer und somit von nur einem Anwender genutzt. Deshalb ist die Wahrscheinlichkeit, dass ein Konflikt auftritt, sehr gering. Außerdem wurde in diesem Fall das *home use* Szenario verwendet, in dem ein Computer im Büro des Anwenders mit einem Computer zu Hause synchro-

[2] siehe http://www.sun.com/
[3] siehe http://www.sun.com/software/solaris/index.jsp

nisiert wird (vgl. Heidemann u. a. 1992, Kapitel 4), so dass nur ein Computer zur ZZeit vom Anwender genutzt werden konnte.

Während einer ersten Testphase wurden einige Probleme festgestellt, da die Synchronisation in erster Linie über langsame Modemleitungen erfolgte (vgl. Heidemann u. a. 1992, Kapitel 3). Ein großes Problem war, dass während der Synchronisation immer alle Dateien verglichen wurden. Als Optimierung, um die Anzahl der Vergleiche zu reduzieren, wurde die Zeit der letzten Synchronisation gespeichert, so dass nur noch alle Dateien verglichen werden mussten, die nach diesem Zeitpunkt verändert wurden (vgl. Heidemann u. a. 1992, Kapitel 3).

Während der Synchronisation, die periodisch erfolgt, werden die veränderten Dateien zwischen den einzelnen Servern übertragen. Zunächst wird die neue Version als temporäre Datei erzeugt. Ist diese neue Version vollständig übertragen worden, kann die ältere Version ersetzt werden. Auf diese Weise ist immer eine vollständige Datei für den Anwender verfügbar (vgl. Guy u. a. 1990, S. 6).

Wie auch *Coda* repliziert *Ficus* Volumes. Die Volumes werden bei Bedarf dynamisch verfügbar gemacht. Dieses als *grafting* (Pfropfen) (vgl. Guy u. a. 1990, S. 7) bezeichnete Verfahren bindet die Volumes an definierten Positionen, sogenannten (*graft points*), in den Verzeichnisbaum ein. Ein *graft point* ist einem Verzeichnis ähnlich, enthält aber Informationen darüber, mit welchen anderen Systemen dieses Volume zu replizieren ist.

Ficus synchronisiert in einer Peer-to-Peer Umgebung, wobei sich die einzelnen Peers kennen.

Aufgrund der Implementierung als Treiber im Betriebssystemkern, war die Portabilität stark eingeschränkt.

Zusammenfassend lassen sich die folgenden Eigenschaften festhalten:

Synchronisationsrichtung	bidirektional
Konflikterkennung	✓
Konfliktlösung	✓
automatische Synchronisation	✓
Server erforderlich	✗

Tabelle 3.7.: Eigenschaften von *Ficus*

3.8. Ivy

Ivy (vgl. Muthitacharoen u. a. 2002) ist ein log-basiertes Peer-to-Peer Dateisystem. Es verwaltet ein Log (Protokoll), in das jede ausgeführte Operation einer Datei eingetragen wird. Diese Log-Einträge werden in einer *distributed Hash-Table (verteilte Hash-Tabelle)* (vgl. Tanenbaum und van Steen 2008, S. 216 ff.) abgelegt. Die *distributed Hash-Table* ist über mehrere Teilnehmer des Peer-to-Peer Netzwerks verteilt. Jeder Teilnehmer speichert dabei nur einen Teil dieser *Hash-Tabelle*. Alle Log-Einträge sind einem Benutzer zugeordnet und werden digital signiert. Dies ermöglicht es, einzelnen Benutzern den Zugriff auf Dateien zu erlauben beziehungsweise zu entziehen.

Bei einem Zugriff auf eine Datei wird in der *distributed Hash-Table* nach der gewünschten Datei gesucht. Anschließend wird aus den Log-Einträgen die Datei rekonstruiert. Um die Zugriffe zu beschleunigen, werden periodisch *Snapshots* – also eine Momentaufnahme des aktuellen Zustands – von Teilen der *distributed Hash-Table* erzeugt und lokal zwischengespeichert. Dieser *Snapshot* enthält in erster Linie Dateien, auf die der jeweilige Anwender zugegriffen hat, so dass dieser *Snapshot* einen Cache darstellt.

Eine Synchronisation ist nur indirekt möglich, da einzelne Blöcke der *distributed Hash-Table* lokal vorgehalten werden. Der Anwender hat jedoch keine Möglichkeit, darauf Einfluss zu nehmen. Wenn der Anwender seinen Computer – beispielsweise ein Notebook – vom Netzwerk trennt, weiß der Anwender nicht, ob der *Snapshot* aktuell genug ist und die gewünschten Dateien in der aktuellen Version auf dem Computer verfügbar sind.

Ivy implementiert das *NFS* (vgl. Network File System) Protokoll und wird entsprechend als lokaler *NFS*-Server eingebunden. Dieser *NFS*-Server ist nur vom lokalen Computer aus erreichbar und dient nur dazu, dem Anwender eine Dateisystem-Schnittstelle für den Zugriff auf die *distributed Hash-Table* zur Verfügung zu stellen.

Durch die Verwendung von Versionsvektoren für die einzelnen Log-Einträge lassen sich Konflikte leicht erkennen. Dem Anwender steht zudem eine Anwendung zur Verfügung, um erkannte Konflikte zu lösen.

Ivy verwendet eine Peer-to-Peer Architektur, um Dateien in einer *distributed Hash-Table* abzulegen. Änderungen werden von *Ivy* nur an ein Protokoll angehängt, so dass keine alten Änderungen verlorengehen. Aufgrund dieser Art der Speicherung müssen einige Daten – wie beispielsweise die aktuelle Dateigröße – bei einer Abfrage berechnet und nicht einfach ausgelesen werden. Dies wirkt sich entsprechend auf die Zugriffsgeschwindigkeit aus.

Die Synchronisation erfolgt indirekt dadurch, dass das Protokoll verwendeter Dateien lokal zwischengespeichert wird. Dadurch kann im Prinzip auch mit Dateien gearbeitet werden, wenn nicht alle Peers der *distributed Hash-Table* verfügbar sind. Allerdings hat der Anwender keine Möglichkeit, diesen Cache zu beeinflussen.

Zusammenfassend lassen sich die folgenden Eigenschaften festhalten.

Synchronisationsrichtung	bidirektional
Konflikterkennung	✓
Konfliktlösung	✓
automatische Synchronisation	✓(nicht beeinflussbar)
Server erforderlich	✗

Tabelle 3.8.: Eigenschaften von *Ivy*

3.9. Zusammenfassung

Abschließend sollen alle vorgestellten Anwendungen, die auf sehr unterschiedliche Arten eine Synchronisiation ermöglichen, verglichen und ihre Vor- und Nachteile dargestellt werden.

Anwendung	Synchronisationsrichtung	Konflikterkennung	Konfliktlösung	automatische Synchronisation	Server erforderlich
Aktenkoffer	bidirektional	✓	✗	✗	✗
rsync	unidirektional	✗	✗	✗	✗
Unison	bidirektional	✓	✓	✗	✗
CVS/ Subversion	bidirektional	✓	✓	✗	✓
Mercurial	bidirektional	✓	✓	✗	✗
Coda	bidirektional	✓	✓	✓	✓(dediziert)
Ficus	bidirektional	✓	✓	✓	✗
Ivy	bidirektional	✓	✓	✓(nicht beeinflussbar)	✗

Tabelle 3.9.: Eigenschaften aller vorgestellten Anwendungen

Der *Windows Aktenkoffer* eignet sich nur bedingt zur Synchronisation von Dateien, da der Inhalt des *Aktenkoffers* mit lokal verfügbaren Dateien synchronisiert wird. Diese Art der Synchronisation erfordert, dass einerseits die Originaldateien nicht verschoben werden und andererseits der *Aktenkoffer* auf andere Computer kopiert wird.

Anwendungen wie *rsync* oder *Unison* eignen sich sehr gut, um Bilder oder digitalisierte Musik zu synchronisieren, da es in so einem Fall nicht notwendig ist, jede Version einer Datei zugänglich zu machen. Bei digitalisierter Musik werden in der Regel nur einige Meta-Daten wie Künstler, Titel oder Album verändert. Relativ wenig Informationen also, die sich auch leicht wieder herstellen lassen, weil sie beispielsweise auf der CD enthalten sind.

Versionskontrollsysteme wie *CVS*, *Subversion* oder *Mercurial* sind in erster Linie entstanden, um verteiltes Entwickeln von Software einfacher handhabbar zu machen. Daher sind sie besonders daraufhin optimiert, einfache Textdateien effizient zu verwalten. Daher eignen sie sich eher, um Quelltext oder andere Textdateien –wie zum Beispiel LaTeX Dokumente – zu verwalten. Bei Dokumenten, die sich relativ häufig ändern, ist der Einsatz einer Versionsverwaltung sinnvoll, da so Änderungen nicht verloren gehen und ein flexibles Arbeiten unterstützt wird. *Mercurial* und andere verteilte Versionskontrollsysteme wie *GIT*[4] oder *Bazaar*[5] sind eine relativ neue Entwicklung im Vergleich zu den älteren, zentralisierten Systemen wie *CVS* oder *Subversion*. Die Peer-to-Peer Architektur ermöglicht dabei eine einfachere Erweiterbarkeit, da aus einem vorhandenen Verzeichnis direkt ein *Repository* und eine *Sandbox* erzeugt werden kann. Es muss also nicht zunächst ein *Repository* erzeugt, alle Dateien eingestellt und eine *Sandbox* erzeugt werden.

Coda als verteiltes Dateisystem ermöglicht eine größtenteils automatische Synchronisation, erfordert dafür aber auch einen dedizierten Server. Daher ist *Coda* eher für größere Installationen geeignet, wenn der erhöhte Aufwand für die Administration des Servers wengier stark ins Gewicht fällt.

Ficus' Einsatzgebiet wäre vermutlich eher eine größere Installation, jedoch konnte *Ficus* nicht getestet werden. Einerseits wurde das Projekt nicht weiter verfolgt, zum anderen existiert keine aktuelle Version, die auf aktuellen Systemen lauffähig wäre.

Ivy bietet interessante Konzepte, gerade das signierte Ablegen aller Dateien. Da nicht festgelegt werden kann, welche Dateien immer im lokalen Cache verfügbar sind,

[4] Linus Torvalds und Junio C Hamano. *GIT*. URL: `http://git.or.cz/` (besucht am 23. 09. 2008).

[5] Canonical Ltd. and Bazaar development community. *Bazaar*. URL: `http://bazaar-vcs.org/` (besucht am 23. 09. 2008).

erscheint der Einsatz wenig sinnvoll, wenn ausgewählte Dateien synchronisiert werden sollen.

4. Szenario

In diesem Kapitel soll anhand von drei Beispielumgebungen beschrieben werden, wie die in Kapitel 3 (S. 22) vorgestellten Anwendungen einzusetzen sind. Anschließend sollen Vor- und Nachteile aufgezeigt werden. Schließlich werden Anforderungen an ein neu zu entwickelndes System definiert werden.

Beispiel 5 *Viele Menschen besitzen ein Mobiltelefon und ein Festnetztelefon. Standardmäßig ist es heute möglich, in beiden ein Telefonbuch zu pflegen. Eine Synchronisation der beiden Geräte ist jedoch in der Regel nicht vorgesehen, so dass die Daten doppelt oder gar nicht gepflegt werden. Im Ergebnis hat der Anwender zwei partiell gepflegte Telefonlisten. Zur Sicherheit wird er vermutlich noch ein drittes Adressbuch pflegen, das als Hauptaddressbuch alle aktuellen Daten enthält.*

Beispiel 6 *Ebenso kann der Fall eintreten, dass ein Anwender mehr als einen Computer besitzt, etwa einen Desktop-PC und ein Notebook. Sobald ein Anwender mehr als einen Computer einsetzt, stellt sich die Frage, wie die Daten zwischen den verschiedenen Computern synchronisiert werden sollen. Denn für viele Aufgaben ist ein Desktop-PC besser geeignet, während für andere Aufgaben ein Notebook die bessere Wahl ist. Es soll also auf keines der beiden Geräte verzichtet werden.*

Vielfach tritt dieses Problem sicherlich nicht als fehlende Synchronisation *zu Tage, stattdessen stellt der Anwender fest, dass ihm beim Arbeiten einige Dateien fehlen, die sich auf dem jeweils anderen Computer befinden. Je häufiger dieser Zustand eintritt und je mehr diese Situation den Anwender stört, desto eher wird ihm dies als fehlende Synchronisation bewusst werden.*

Eine einfache Möglichkeit, diese Problematik etwas zu verlagern, ist beispielsweise der Einsatz einer Network Attached Storage (NAS)*, einer netzwerkfähigen, externen Festplatte. Dann existieren allerdings keine zwei Geräte mehr, die synchronisiert werden müssten, sondern drei. Im Idealfall wird ein* NAS *als Synchronisationspunkt eingesetzt, der als zentraler Speicher die aktuelle Versionen der Dateien enthält. Diese netzwerkfähige, externe Festplatte wird somit als Server verwendet.*

Beispiel 7 *Ebenso tritt häufig der Fall ein, dass Mitarbeiter in Firmen ein Notebook als Arbeitsplatzrechner erhalten, um ihnen eine hohe Flexibilität zu ermöglichen. Gerade in Firmen hat diese Art der Flexibilität einen großen Nachteil: Die Notebooks sind bei Mitarbeitern, die häufig außer Haus sind, nur zeitweise und größtenteils nur tagsüber im Firmennetzwerk erreichbar. Auf Computern, die sich permanent in den Räumen der Firmen und somit im Firmennetzwerk befinden, sollten in der Regel keine Dateien der Anwender abgelegt werden, weil die Speicherung und Verwaltung die Aufgabe zentraler Server ist. Im Gegensatz dazu enthalten Notebooks Dateien der Anwender, da diese außerhalb der Firma keine andere Möglichkeit der Dateiablage haben. Wiederkehrende Aufgaben, wie eine nächtliche Datensicherung, sind somit so gut wie unmöglich. In solchen Fällen muss der Mitarbeiter die eigenen Dateien regelmäßig auf Dateiserver kopieren, wenn diese Dateien in der Datensicherung berücksichtigt werden sollen. Dieses Vorgehen ist jedoch fehleranfällig und aufwändig, da es Aufgabe des Anwenders ist, die Synchronisation regelmäßig durchzuführen.*

Falls wie in Beispiel 5 (S. 43) zwei Telefonbücher zu synchronisieren sind, existieren möglicherweise Lösungen, beispielsweise mit Hilfe einer Anwendung auf einem Computer und dem Standard *SyncML* (vgl. Open Mobile Alliance 2008). In der Regel lassen sich – zumindest unter *Microsoft Windows* oder *Apple MacOS X* – Adressbücher von Mobiltelefonen recht problemlos mit Anwendungen auf dem Computer verwalten und synchronisieren. Diese Möglichkeiten sollen hier jedoch nicht betrachtet werden, weil der Schwerpunkt in dieser Arbeit auf der Synchronisation von Dateien (Dokumente, Bilder, digitalisierte Musik, . . .) liegt.

Alle in diesem Kapitel beschrieben Anwendungsfälle, bei denen Dateien zu synchronisieren sind, lassen sich mit den in Kapitel 3 (S. 22) beschriebenen Anwendungen – mit Ausnahme von *Ficus* und *Ivy* –umsetzen. Bis auf *Ficus* und *Ivy*, die sich nicht verwenden ließen, wurden alle in Kapitel 3 (S. 22) vorgestellten Anwendungen in der Praxis getestet. Jedoch sind die verschiedenen Anwendungen unterschiedlich gut geeignet. Für die Synchronisation digitialisierter Musik oder Bilder eigenet sich *Unison* am besten und für Dokumente – insbesondere Textdokumente[1] – ist der Einsatz eines Versionskontrollsystems wie *CVS*, *Subversion* oder *Mercurial* sinnvoll. Jedoch ist in allen Fällen keine automatische Synchronisation möglich. Der Anwender muss in jedem Fall die zu synchronisierenden Dateien auswählen und die Synchronisation initiieren.

Coda erlaubt zwar, die Synchronisation zu automatisieren, jedoch erfordert dies den Betrieb eines dedizierten Servers. Dadurch ergibt sich ein erhöhter Aufwand,

[1] *OpenOffice.org Writer* und *Microsoft Word* Dateien sind keine Textdokumente, da sie als Binärdateien abgelegt werden.

der umso höher ausfällt, je weniger Computer zu synchronisieren sind. Für Firmen ist dies eine relativ einfache Möglichkeit Daten zu synchronisieren, zumal dort der Betrieb eines dedizierten Servers kein Problem darstellt. Lediglich die Integration der Benutzerverzeichnisse erschwert unter Umständen eine Nutzung von *Coda*. Für den privaten Anwender ist dieser Aufwand sicherlich nicht akzeptabel. Außerdem sollte ein dedizierter Server auch entsprechend nutzbar, also möglichst permanent in Betrieb sein. In Firmen ist ein permanenter Betrieb unumgänglich, für den privaten Bereich in der Regel jedoch nicht akzeptabel.

4.1. Lösungsansatz

Aus diesen Gründen wäre eine einfach zu nutzende und vor allem automatisch im Hintergrund ablaufende Synchronisation hilfreich. Alle Dateien innerhalb eines bestimmten Verzeichnisses werden automatisch mit anderen gewünschten Computern synchronisiert. Die Synchronisation erfolgt automatisch mit den erreichbaren Computern.

Daraus ergeben sich folgende funktionale Anforderungen.

Synchronisationsrichtung	bidirektional
Konflikterkennung	✓
Konfliktlösung	✓
automatische Synchronisation	✓
Server erforderlich	✗

Tabelle 4.1.: Eigenschaften der neu zu entwickelnden Anwendung

Außerdem lassen sich noch folgende nicht-funktionale Anforderungen festhalten.

- Alle Computer beziehungsweise Verzeichnisse, mit denen eine Synchronisation erfolgen soll, sind bekannt. Eine Lokalisierung der beteiligten Computer ist daher nicht notwendig.

- In der Regel wird jedes Verzeichnis nur mit einigen wenigen Verzeichnissen synchronisiert werden. Das Peer-to-Peer Netzwerk, das sich zwischen allen zu synchronisierenden Verzeichnissen bildet, wird eher klein sein und nicht aus hunderten oder tausenden von Teilnehmern bestehen.

- Eine Konflikterkennung und -lösung muss vorhanden sein, jedoch sollten Konflikte relativ selten auftreten, weil nur ein einziger Anwender mit den Dateien arbeitet.

Ein mögliches Konzept, das die genannten Anforderungen erfüllt, wird im nächsten Kapitel vorgestellt.

5. Konzeption des synchronisierenden Peer-to-Peer Dateisystems

In diesem Kapitel soll das Konzept eines synchronisierenden Peer-to-Peer Dateisystems vorgestellt werden. Zunächst soll die Verwendung des Systems aus Sicht des Anwenders betrachtet werden. Anschließend wird auf die Synchronisation, die Konflikterkennung und die Konfliktlösung im Allgemeinen eingegangen, bevor eine spezielle Betrachtung in Bezug auf die Synchronisation mit Hilfe des Dateisystems erfolgt.

Zunächst wird eine Umgebung dargestellt, in der das synchronisierende Peer-to-Peer Dateisystem eingesetzt werden könnte. Abbildung 5.1 (S. 47)[1] stellt ein Beispiel dar, wie eine Synchronisation zwischen unterschiedlichen Geräten erfolgen könnte.

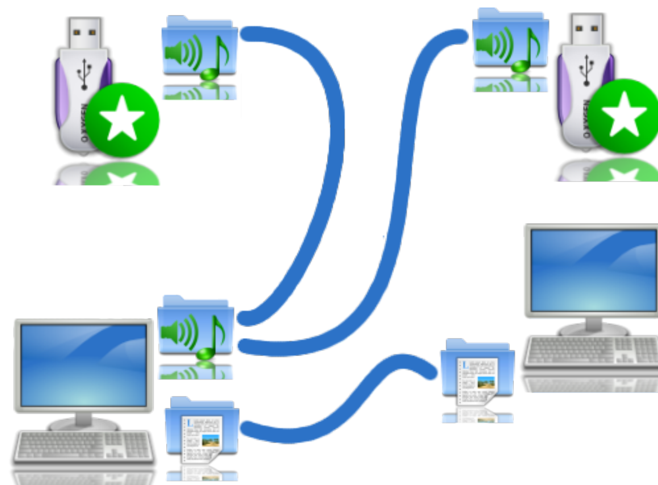

Abbildung 5.1.: Architektur und Einsatz verschiedener synchronisierender Peer-to-Peer Dateisysteme

Die Linien zwischen den Verzeichnissymbolen zeigen an, welche Verzeichnisse miteinander synchronisiert werden. In diesem Beispiel werden Dokumente zwischen

[1] Magnus Bjureblad. *Reflective Icons - Oxygen*. Lizenz GPL. URL: http://www.kde-look.org/content/show.php/?content=77300 (besucht am 09. 09. 2008).

beiden Computern synchronisiert. Digitalisierte Musik des einen Computers wird zudem noch mit zwei externen Datenträgern (USB-Sticks) synchronisiert. In allen Fällen soll die Synchronisation *bidirektional und automatisch* erfolgen.

Aus Sicht des Anwenders weisen einige Verzeichnisse die besondere Eigenschaft auf, dass sie die enthaltenen Objekte mit anderen Verzeichnissen, die ebenfalls diese Eigenschaft besitzen, synchronisieren. Der Anwender soll mit diesen speziellen Verzeichnissen wie mit jedem anderen Verzeichnis arbeiten können. Zu gewissen Zeitpunkten werden allerdings im Hintergrund automatisch alle enthaltenen Objekte synchronisiert. Es wird dabei versucht, Konflikte zu vermeiden oder auftretende Konflikte, wenn möglich, automatisch zu lösen. Die Kommunikation zwischen diesen Verzeichnissen während der Synchronisation kann sowohl lokal – wie im Falle der externen Datenträger –, als auch über ein Netzwerk – wie im Falle des anderen Computers – erfolgen. Da jedes Verzeichnis mit jedem anderen Verzeichnis synchronisiert werden kann, bilden die verschiedenen Verzeichnisse eine Peer-to-Peer Architektur. Die Abbildung 5.2 (S. 48) zeigt die Topologie des Beispiels aus Abbildung 5.1 (S. 47).

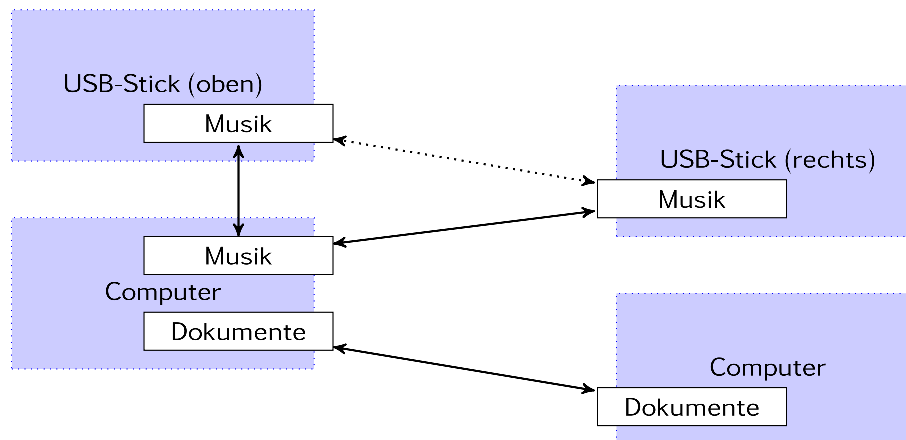

Abbildung 5.2.: Logische Topologie des Beispiels aus Abbildung 5.1 (S. 47)

In der topologischen Darstellung werden die Computer und die externen Datenträger durch die jeweiligen eingefärbten Felder dargestellt. Die Synchronisation erfolgt zwischen den Knoten, die jeweils entsprechend ihres Inhalts beschriftet sind, und wird durch die Kanten dargestellt. Die beiden Darstellungen unterscheiden sich in einem wichtigen Punkt: die beiden Musik-Verzeichnisse auf den externen Datenträgern sind ebenfalls mit einer gepunkteten Kante verbunden. Es erfolgt *indirekt* eine Synchronisa-

tion zwischen diesen beiden Verzeichnissen — *indirekt*, da der Computer die beiden externen Datenträger als eine Art „Vermittler“ verbindet.

5.1. Synchronisation

Um nun eine automatische Synchronisation zu ermöglichen, können verschiedene Ansätze verfolgt werden. Ein sehr einfacher und naheliegender Ansatz ist, dass das Synchronisationsprogramm periodisch bestimmte und vom Benutzer ausgewählte Verzeichnisse auf Veränderungen durchsucht und bei Bedarf eine Synchronisation startet.

Es kann beispielsweise für jede Datei und jedes Verzeichnis der Zeitpunkt der letzten Änderung gespeichert werden. Werden nun Dateien oder Verzeichnisse während der periodischen Überprüfung gefunden, deren Zeitpunkt der letzten Modifikation neuer als der gespeicherte Zeitpunkt ist, so wurde die Datei vermutlich verändert. Das Speichern einer unveränderten Datei aktualisiert auch die letzte Modifikationszeit einer Datei, so dass ohne Vergleich des Inhalts nicht entschiedenen werden kann, ob tatsächlich eine Veränderung stattgefunden hat. Um dies entscheiden zu können, könnte zusätzlich noch der kryptografische Hash-Wert des Dateiinhalts der entsprechenden Dateien gespeichert werden. Damit lässt sich dann entscheiden, ob eine Datei verändert wurde und eine Synchronisation erforderlich ist (siehe Kapitel 2.1.2 (S. 8)).

Die Synchronisation erfolgt sodann durch Kopieren der veränderten Dateien. Dazu könnte beispielsweise auf *rsync* (siehe Kapitel 3.2 (S. 23)) zurückgegriffen werden, um möglichst effizient die Dateien zu kopieren.

Nachteilig bei einer Anwendung, die periodisch bestimmte Verzeichnisse auf Veränderungen durchsucht, ist genau dieses periodische Durchsuchen. Dies erzeugt eine unnötige Belastung des Systems, wodurch beispielsweise bei Notebooks die Akkulaufzeit reduziert wird. Wird die Zeitspanne zwischen zwei Suchläufen vergrößert, reduziert dies zwar diese Belastung, dafür dauert möglicherweise die Synchronisation entsprechend länger, da mehr veränderte Dateien vorliegen. Außerdem steigt die Dauer des Durchsuchens mit der Anzahl der Verzeichnisse und Dateien, die zu synchronisieren sind. Als Optimierungsmöglichkeit zur Verringerung der Belastung beim periodischen Durchsuchen bietet sich unter *Linux* beispielsweise *inotify* (vgl. inotify) an. *inotify* erlaubt es, dass sich eine Anwendung benachrichtigen lässt, wenn zuvor registrierte Dateien oder Verzeichnisse verändert werden. Dadurch kann das periodische Durchsuchen etwas reduziert werden, da nur seit der letzten Synchronisation veränder-

te Dateien beziehungsweise Verzeichnisse überprüft werden müssten. Dennoch müssen immer noch Dateien und Verzeichnisse periodisch untersucht werden.

Nachdem die Synchronisation vorgestellt wurde, wird im nächsten Abschnitt die Erkennung von Konflikten betrachtet.

5.2. Konflikterkennung

Da die Dateien und Verzeichnisse bidirektional synchronisiert werden sollen, können Konflikte auftreten. Diese müssen dann sicher erkannt werden, um eine Konfliktlösung zu ermöglichen.

Zunächst soll jedoch erläutert werden, wie ein Konflikt erkannt werden kann. In Kapitel 2.2.2 (S. 18) wurde bereits auf diese Problematik eingegangen und anhand einiger Beispiele verdeutlicht.

Ein Konflikt tritt nur während einer Synchronisation zwischen zwei Synchronisationspartnern auf. Es existieren jedoch vielfältige Möglichkeiten, einen Konflikt zu erzeugen. Diese lassen sich in folgender Definition erfassen:

Definition 7 *Sei p_1 der Pfad des einen Synchronisationspartners, der mit dem Pfad p_2 des anderen Synchronisationspartners synchronisiert wird, und seien beide Pfade gemäß Definition 1 (S. 3) identisch.*

Sei t_S der Zeitpunkt der Synchronisation und $d(p)$ eine Funktion, die den Wert 1 ergibt, falls der Pfad p (Datei oder Verzeichnis) vor dem Zeitpunkt t_s verändert wurde, andernfalls den Wert 0, dann tritt ein Konflikt auf, wenn gilt:

$$\begin{aligned} d(p_1) &= 1 \qquad \wedge \\ d(p_2) &= 1 \end{aligned}$$

Also nur wenn beide *Pfade verändert wurden, tritt ein Konflikt auf.*

Um zu erkennen, ob ein Pfad verändert wurde, kann bei Dateien der kryptografische Hash-Wert berechnet werden. Zusätzlich sollte noch der Zeitpunkt der letzten Modifikation gespeichert werden, um nur für veränderte Dateien den kryptografischen Hash-Wert neu zu berechnen. Die Speicherung des Zeitpunkts stellt kein Problem dar, da für die Speicherung und den Vergleich immer nur die lokale Zeit verwendet wird. Neu hinzugekommene und gelöschte Dateien oder Verzeichnisse lassen sich ebenfalls erkennen, indem die gespeicherten Informationen zu den einzelnen Dateien bezie-

hungsweise Verzeichnissen mit den jeweiligen Pfaden des Dateisystems verglichen werden (siehe auch Kapitel 2.2.2 (S. 18)).

Es ist jedoch nicht ausreichend, nur den Dateiinhalt beim Vergleich zu betrachten. Ebenso können die Berechtigungen von Dateien oder Verzeichnissen verändert werden. Berücksichtigt die Funktion $d(p)$ aus Definition 7 (S. 50) diese ebenfalls, dann wird auch ein Konflikt erkannt, wenn der eine Synchronisationspartner den Dateiinhalt und der andere die Berechtigungen verändert hat. Ein Anwender sieht dies vermutlich nicht unbedingt als Konflikt an, da zwar beide Dateien verändert wurden, die Änderungen sich aber nicht gegenseitig beeinflussen. Es könnten also beide Änderungen in beliebiger Reihenfolge ausgeführt werden, ohne dass die Dateien der Synchronisationspartner anschließend unterschiedlich wären.

Wenn die Anzahl der Synchronisationspartner steigt, wird nicht unbedingt eine Synchronisation zwischen allen Synchronisationspartnern erfolgen. Zudem muss der Anwender die Verbindungen zwischen den Synchronisationspartnern manuell herstellen. Beispielsweise könnte die Synchronisation zwischen einer externen Festplatte und einem Notebook erfolgen. Diese externe Festplatte könnte auch noch mit weiteren Computern synchronisiert werden. Diese Variante könnte genutzt werden, wenn die Computer und das Notebook nicht direkt miteinander verbunden werden können. Durch solche Konstellationen entstehen Ketten zwischen den Synchronisationspartnern. Diese Ketten ermöglichen das Auftreten von Konflikten zwischen Synchronisationspartnern, bei denen diese Konflikte nicht erwartet werden. Dies soll durch das Beispiel 8 (S. 51) verdeutlicht werden.

Beispiel 8 *Wird eine Kette von Synchronisationspartnern aufgebaut, die alle ein Verzeichnis synchronisieren, können Konflikte an nicht sofort ersichtlichen Stellen auftreten. Die Abbildung 5.3 (S. 52) stellt zunächst ein solches Szenario dar. Alle Geräte synchronisieren beispielsweise das Verzeichnis* `~/docs`. *Die gepunkteten Pfeile zeigen an, welche Geräte jeweils untereinander das Verzeichnis synchronisieren. Zwischen* allen *Geräten, die nicht mit einer Kante verbunden sind, erfolgt eine* indirekte *Synchronisation wie dies auch in Abbildung 5.2 (S. 48) schon beispielhaft gezeigt wurde. In diesem Beispiel wird die Datei* `letter.tex` *sowohl auf dem* USB-Stick 1 *als auch auf dem* Firmen Notebook *ohne zwischenzeitliche Synchronisation verändert, so dass diese Änderungen in Konflikt zueinander stehen. Dies wird durch den hervorgehobenen Pfeil zwischen diesen beiden Geräten verdeutlicht. Dieser Konflikt tritt jedoch zunächst nicht während der Synchronisation auf, sondern ist nur erkennbar, wenn das* gesamte Peer-to-Peer Netzwerk *betrachtet wird.*

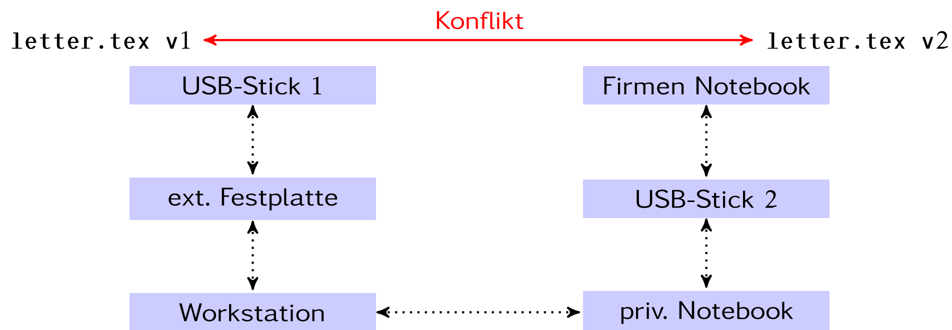

Abbildung 5.3.: Verkettung mehrerer synchronisierender Verzeichnisse und Änderung der Datei `letter.tex` bei zwei Teilnehmern

In Abbildung 5.4 (S. 52) ist dargestellt, dass zunächst die jeweiligen Änderungen konfliktfrei synchronisiert werden können, da der Konflikt nur bei Betrachtung des gesamten Peer-to-Peer Netzwerks *ersichtlich ist; dies ist durch die hervorgehobenen Pfeile angedeutet. Erst wenn die* Workstation *und das* private Notebook *miteinander synchronisiert werden, wird der Konflikt tatsächlich erkannt und der Anwender kann diesen lösen. Die Synchronisation wird durch die gepunkteten Pfeile dargestellt. Auch wenn die Darstellung vermuten lässt, dass die Synchronisation direkt nach Erhalt der neuen Version erfolgt, so ist dies nicht der Fall. Nach dem Erstellen der beiden neuen Versionen kann durchaus ein längerer Zeitraum verstreichen, bevor der Konflikt auftritt.*

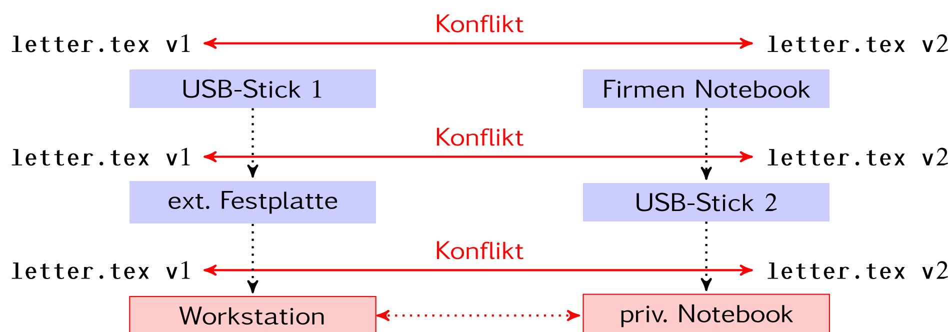

Abbildung 5.4.: Verzögertes Auftreten des Konflikts erst bei der Synchronisation zwischen *Workstation* und *privatem Notebook*

Nachdem der Konflikt zwischem der Workstation *und dem* privaten Notebook *gelöst wurde, wird die daraus resultierende neue Version dieser Datei (*`letter.tex v3`*) entsprechend wieder synchronisiert, so dass schlussendlich auch der* USB-Stick 1 *und das* Firmen Notebook *die nun gültige Version enthalten (siehe Abbildung 5.5 (S. 53)). Auch diese*

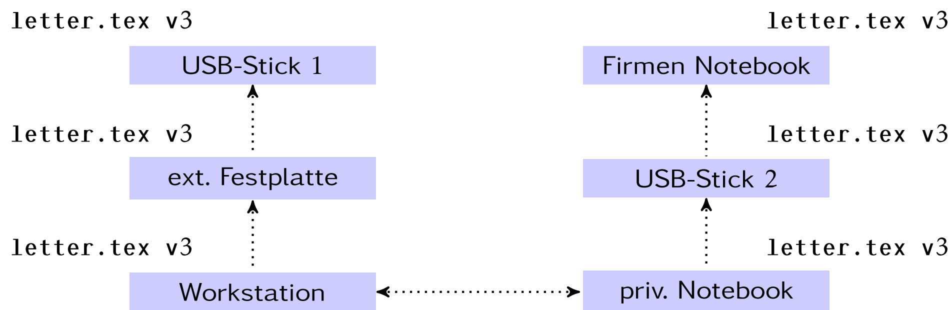

Abbildung 5.5.: Synchronisation der neuen Version `letter v3` nach Auflösung des Konflikts

Synchronisation der neuen Version kann durchaus über einen längeren Zeitraum geschehen, so dass unter Umständen durch diese Synchronisation erneut ein Konflikt – auch an anderer Stelle – auftreten kann. Dies hängt davon ab, wie häufig die einzelnen Verzeichnisse synchronisiert werden und aus wie vielen Teilnehmern dieses Peer-to-Peer Netzwerk besteht.

Je häufiger eine Synchronisation erfolgt, desto seltener sollten der gerade beschriebene Effekt auftreten.

Der in Beispiel 8 (S. 51) beschriebene Effekt, dass Konflikte zu einem späteren Zeitpunkt und zwischen unerwarteten Synchronisationspartnern auftreten, ließe sich umgehen, indem alle nicht verbundenen Synchronisationspartner automatisch verbunden werden. Jedoch könnte dies zu vom Anwender unerwünschten Verbindungen führen. Außerdem ist nicht ersichtlich, ob eine direkte Synchronisation zwischen allen Synchronisationspartnern möglich ist.

Wurde nun ein Konflikt erkannt, so muss dieser behoben werden, damit diese Datei oder das Verzeichnis weiterhin synchronisiert werden kann. Welche Möglichkeiten der Konfliktlösung existieren und inwieweit der Anwender mit einbezogen werden muss, wird im nächsten Kapitel betrachtet.

5.3. Konfliktlösung

Wenn ein Konflikt während der Synchronisation erkannt wird, darf die entsprechende Datei oder das entsprechende Verzeichnis nicht aktualisiert werden, da andernfalls Änderungen überschrieben werden könnten und somit verloren gingen. Der erkannte Konflikt kann dann einer von zwei Kategorien zugeordnet werden:

1. automatisch lösbar

2. nur manuell lösbar

Ein Konflikt ist *automatisch lösbar*, wenn der Anwender nicht nach einer Lösung gefragt werden muss. Es kann also direkt entschieden werden, wie zu verfahren ist, um die Änderungen zusammenzuführen, so dass keine Änderungen verloren gehen und die Datei oder das Verzeichnis weiterhin synchronisiert werden kann.

Wurden beispielsweise bei einem Synchronisationspartner die Zugriffsberechtigungen einer Datei verändert und bei einem anderen der Dateiinhalt, so ist der Konflikt *automatisch lösbar*. Es können beide Änderungen durchgeführt werden, ohne den Anwender fragen zu müssen und ohne, dass Änderungen verloren gehen. Nach Lösung des Konflikts enthalten beide Synchronisationspartner die Datei mit dem neuen Dateiinhalt und den angepassten Zugriffsberechtigungen.

Im Gegensatz dazu muss bei einem *nur manuell lösbaren* Konflikt der Anwender entscheiden, wie der Konflikt zu lösen ist. Ein solcher Konflikt tritt beispielsweise auf, wenn bei beiden Synchronisationspartnern der Dateiinhalt verändert wurde. Es kann dann nicht mehr *automatisch* entschieden werden, welcher Dateiinhalt zukünftig verwendet werden soll. Der Anwender hat nun drei verschiedene Möglichkeiten der Konfliktlösung zur Auswahl:

1. Der Anwender verwendet die Version des einen Synchronisationspartners und verwirft die andere Version.

2. Entsprechend der vorherigen Möglichkeit, nur wird die Version des anderen Synchronisationspartners verwendet.

3. Der Anwender führt beide Versionen zusammen und nimmt diese als Folgeversion.

Um dem Anwender die Entscheidung für eine der drei Möglichkeiten zu erleichtern, kann ihm als Hilfestellung die Version des jeweiligen Synchronisationspartners angeboten werden. So hat der Anwender beide Versionen im direkten Zugriff, kann diese vergleichen und entscheiden, wie fortzufahren ist. Da beide Versionen der Datei einen identischen Namen besitzen, könnte beispielsweise der Name des Synchronisationspartners als Suffix an den Dateinamen angehängt werden. So ist die Herkunft der Datei klar ersichtlich. Das Listing 5.1 (S. 55) zeigt an einem Beispiel diese Möglichkeit. `test` ist die lokal verfügbare Version der Datei, und `test@workstation` ist die Version des Synchronisationspartners *workstation*.

```
[user@notebook]:~/sync > ls -1
  test
  test@workstation
```

Listing 5.1: Zwei in Konflikt stehende Versionen einer Datei

Ein Vorteil dieser Vorgehensweise ist, dass die Änderungen der jeweiligen Versionen weiterhin synchronisiert werden können. Jedoch muss die Synchronisation kreuzweise erfolgen. Die lokale Version der Datei muss mit der zur Konfliktlösung erzeugten Version des Synchronisationspartners synchronisiert werden. Entsprechend muss auch die andere Version synchronisiert werden. Die Abbildung 5.6 (S. 55) stellt dies übersichtlich dar.

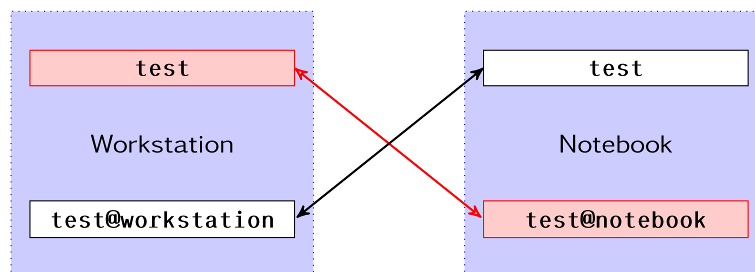

Abbildung 5.6.: Kreuzweise Synchronisation der Versionen

So besteht die Möglichkeit, dass der Anwender beide Versionen – genauer Zweige – der Datei weiter verändern kann, um somit eine Zusammenführung der Zweige zu vereinfachen. Dieses Vorgehen entspricht der Zusammenführung zweier Zweige, wie es beispielsweise von *Mercurial*, das in Kapitel 3.5 (S. 30) vorgestellt wurde, umgesetzt wird.

Hat sich der Anwender nun für eine Möglichkeit entschieden, kann er den Konflikt lösen und dies der Anwendung mitteilen. Diese muss anschließend noch einige Aktionen ausführen, um den Konflikt ebenfalls zu lösen:

1. Dem Synchronisationspartner muss mitgeteilt werden, dass der Konflikt gelöst wurde.

2. Im Zuge der Konfliktlösung muss dem Synchronisationspartner die neue Version der Datei übermittelt werden, die die dort vorhandene Version überschreibt. Dies darf selbstverständlich *keinen* Konflikt auslösen.

3. Wurde dem Anwender die Version des Synchronisationspartners zur Hilfe bei der Konfliktlösung angeboten, sollte sie nach Auflösung des Konflikts gelöscht werden. Dies darf ebenfalls zu keinem Fehler führen, da diese Dateien nur temporär erzeugt wurden.

4. Beide Synchronisationspartner enthalten die neue Version der Datei und der Konflikt wurde gelöst. Fortan wird die neue Version der Datei synchronisiert.

Der Konflikt ist gelöst und die Datei kann wieder problemlos synchronisiert werden.

5.4. Dateisystem

Bisher war es notwendig, die zu synchronisierenden Verzeichnisse periodisch nach Veränderungen zu durchsuchen, um eine Liste der zu synchronisierenden Dateien und Verzeichnisse zu erstellen. Mit Hilfe eines Dateisystems wird dieses periodische Durchsuchen überflüssig, da während der Verwendung bereits alle für eine Synchronisation notwendigen Informationen gesammelt werden können.

Die Implementierung eines neuen Dateisystems zur möglichst effizienten Synchronisation erscheint zunächst viel zu aufwändig. Daher soll als erstes der Begriff des *Dateisystem* im Zusammenhang mit der Synchronisation erläutert und dann im Kontext dieses Konzepts detailliert betrachtet werden.

Die Aufgabe eines Dateisystems ist es, den Zugriff auf Dateien und Verzeichnisse, die zur Strukturierung dienen (vgl. Tanenbaum 2003*b*, S. 421 ff.), zu ermöglichen und die entsprechenden Daten auf Medien zu verwalten (vgl. Tanenbaum 2003*b*, S. 407 ff.). Wie die Dateien und Verzeichnisse auf den Medien abgelegt und wiedergefunden werden, ist Aufgabe des Dateisystems und jedes Dateisystem verwendet eigene Techniken (vgl. Tanenbaum 2003*b*, S. 428, S. 461 ff.), die auch von der Art des Mediums (Festplatte, CD-ROM, DVD, Flash-Speicher, . . .) abhängen.

Weiterhin besitzt ein Dateisystem eine Schnittstelle, die indirekt von verschiedenen Anwendungen genutzt wird, um dem Anwender die Verwaltung von Dateien und Verzeichnissen zu ermöglichen. Dazu zählen Anwendungen, um sich beispielsweise den Inhalt des Dateisystems anzeigen zu lassen, Verzeichnisse zu erstellen, zu löschen oder Dateien zu kopieren und umzubenennen. Diese Anwendungen arbeiten in der Regel unabhängig vom verwendeten Dateisystem, auf dem die Dateien und Verzeichnisse abgelegt sind. Ein Anwender kommt also in der Regel nicht direkt mit dem Dateisystem

in Berührung, sondern benutzt es nur durch Verwendung verschiedener Anwendungen (vgl. Tanenbaum und van Steen 2008, S. 415 ff., S. 426 ff.).

Das hier zu entwickelnde Dateisystem soll die Verwaltung von Dateien und Verzeichnissen *nicht* implementieren. Es existieren bereits viele verschiedene Dateisysteme, die diese Verwaltung implementieren und zuverlässig funktionieren, wie beispielsweise das unter Linux verbreitete *ext2* beziehungsweise dessen Nachfolger *ext3* (vgl. Tanenbaum 2003*b*, S. 796) oder unter *FreeBSD UFS2* (vgl. Lehey 2006, S. 190 ff.).

Das in diesem Konzept betrachtete synchronisierende Dateisystem gliedert sich zwischen der typischen Dateisystem-Schnittstelle und einem existierenden Dateisystem ein. Für alle Anwendungen bleibt der bisherige Zugriff erhalten, so dass diese weiterhin ohne Anpassung eingesetzt werden können. Weiterhin entfällt auch die Implementierung der Verwaltung der Dateien, da diese Aufgabe delegiert werden kann. Die Abbildung 5.7 (S. 57) verdeutlicht die Integration des synchronisierenden Peer-to-Peer Dateisystems in das System. Die jeweils eingefärbten Schichten sind Teil des synchronisierenden Dateisystems, die anderen Teil des verwendeten Systems, bestehend aus Anwendungen, Betriebssystem und Hardware.

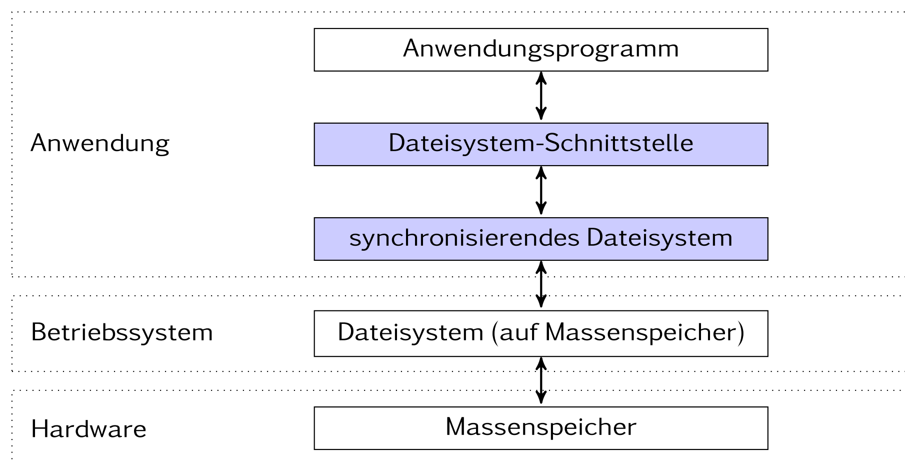

Abbildung 5.7.: Schichtenarchitektur und Systemintegration des synchronisierenden Peer-to-Peer Dateisystems

Die Pfeile dienen dazu, die Richtung und Reihenfolge der Aufrufe zu verdeutlichen. Das *Anwendungsprogramm* greift über die *Dateisystem-Schnittstelle* auf eine Datei zu und verändert diese. Dieser Aufruf wird an das *synchronisierende Dateisystem* weitergereicht, das wiederum den Aufruf an das darunterliegende *Dateisystem (auf Massenspeicher)* delegiert, so dass die Veränderung auf dem entsprechenden *Massenspeicher* persistent

abgelegt wird. Alle anderen Anforderungen, wie etwas das Lesen einer Datei, werden entsprechend behandelt, so dass das *synchronisierende Dateisystem* jeden Aufruf mit allen notwendigen Parametern erhält und weiterleiten muss. Die Rückgabewerte der jeweiligen Aufrufe, die in der Regel den Erfolg oder Misserfolg des Aufrufs signalisieren, werden entsprechend an die aufrufende Schicht zurückgeliefert.

In Abbildung 5.8 (S. 58) werden die beiden eingefärbten Komponenten aus Abbildung 5.7 (S. 57) separat betrachtet. Auf der linken Seite dieser Abbildung sind diese Schichten des synchronisierenden Dateisystems als Komponenten dargestellt. Die Dateisystem-Schnittstelle stellt keine wirkliche Komponente dar, da sie nur eine zu implementierende Schnittstelle definiert. Auf der rechten Seite ist die Komponente des synchronisierenden Dateisystems mit den enthaltenen Komponenten dargestellt.

Abbildung 5.8.: Komponenten des synchronisierenden Peer-to-Peer Dateisystems

Die eingefärbten Komponenten stellen die verschiedenen Komponenten innerhalb des synchronisierenden Dateisystems dar. Diese sollen nachfolgend erläutert werden.

Synchronisation: Diese Komponente führt die empfangenen Operationen aus, so dass die lokal vorhandenen Dateien und Verzeichnisse dem Stand des Synchronisationspartners entsprechen.

Protokollierung: In dieser Komponente erfolgt die Protokollierung der einzelnen Operationen, so dass diese für eine spätere Synchronisation verwendet werden können.

Dateisystem: Die Dateisystem-Komponente dient dazu, die Anforderungen der Anwendung auszuführen. Einerseits müssen die Informationen für die *Protokollie-*

rung ermittelt und an diese weitergegeben werden. Andererseits muss der eigentliche *Dateizugriff* auf das Dateisystem, das die Daten auf dem Massenspeicher verwaltet, erfolgen. Diese Komponente leitet die Aufrufe an die entsprechenden Komponenten weiter.

Anwender: Diese Komponente ermöglicht es dem Anwender, mit dem synchronisierenden Dateisystem zu kommunizieren, um beispielsweise Konflikte zu lösen oder die Konfiguration anzupassen.

Dateizugriff: In dieser Komponente erfolgt der Zugriff unter Verwendung der Systemfunktionen auf das Dateisystem, das die Daten auf dem Massenspeicher verwaltet. Diese Komponente kontrolliert ebenfalls den Zugriff auf die Dateien und Verzeichnisse, da ein Zugriff entweder durch die diese oder die Synchronisations-Komponente erfolgen darf, jedoch nie gleichzeitig durch beide.

5.4.1. Synchronisation

In diesem Abschnitt soll betrachtet werden, welche Informationen für eine Synchronisation notwendig sind und wie diese erfolgen kann (siehe Kapitel 2.2.3 (S. 19)). Dazu muss betrachtet werden, welche Informationen protokolliert werden können, so dass sie zu einem späteren Zeitpunkt für eine Synchronisation genutzt werden können. Da die Implementierung die Dateisystem-Schnittstelle verwendet, müssen alle notwendigen Informationen aus den Aufrufen beziehungsweise Operationen gewonnen und persistent abgelegt werden. Dies wird nachfolgend die *Protokollierung der Operationen* genannt.

Anwendungen können eine Vielzahl von Operationen nutzen, um Dateien und Verzeichnisse zu verwalten. Allen gemein ist jedoch, dass auf Dateien und Verzeichnisse immer über einen Namen zugegriffen wird, der aus Sicht der Anwendung und des Anwenders Dateien oder Verzeichnisse eindeutig identifiziert. Fortan soll dieser eindeutige Name als Pfad bezeichnet werden und sowohl Dateien als auch Verzeichnisse einschließen.

In Kapitel 2.1.2 (S. 11) wurden bereits die Operationen, die von einer Dateisystem-Schnittstelle zur Verfügung gestellt werden, in verschiedene Kategorien eingeteilt. Diese Einteilung soll hier kurz wiederholt und anschließend erweitert werden.

Operationen, *die Dateiinhalte oder Verzeichnisse nicht verändern*, sollen hier nicht weiter betrachtet werden, da diese Operation für eine Synchronisation irrelevant sind.

Für eine Synchronisation sind aber die Operationen der beiden anderen Kategorien interessant:

1. Operationen, die Dateiinhalte oder Verzeichnisse verändern
2. Operationen, die Dateiinformationen verändern

Um eine Synchronisation mit Hilfe der Operationen zu ermöglichen, sind einige Vorkehrungen zu treffen, damit das Protokoll und die Datei beziehungsweise das Verzeichnis konsistent sind. Datei beziehungsweise Verzeichnis und Protokoll dürften nicht voneinander abweichen, da der Anwender andernfalls unerwartete Daten nach erfolgter Synchronisation vorfinden könnte. Das Protokollieren und Ausführen der Operation muss daher wie eine Transaktion behandelt werden, entweder beides kann erfolgreich ausgeführt werden oder keine Aktion wird ausgeführt (vgl. Tanenbaum und van Steen 2008, S. 37 ff.; Schill und Springer 2007, S. 97 ff.). Nachfolgend soll beschrieben werden, wie dies erreicht werden kann.

Zunächst muss die Operation vorläufig protokolliert werden, schlägt dies fehl, darf die Operation ebenfalls nicht ausgeführt werden. Kann die Operation erfolgreich protokolliert werden, muss die tatsächliche Operation ausgeführt werden, um den Pfad (Datei oder Verzeichnis) zu verändern. War dies erfolgreich, muss die protokollierte Operation festgeschrieben werden. Konnte die Operation nicht erfolgreich ausgeführt werden, so muss die vorläufig protokollierte Operation wieder entfernt werden. Ein Problem besteht jedoch weiterhin: Wie ist vorzugehen, wenn das Festschreiben der vorläufig protokollierten Operation fehlschlägt? Die bereits ausgeführte Operation kann nicht mehr rückgängig gemacht werden. Auch wenn dieser Fall theoretisch möglich wäre, soll er hier nicht berücksichtigt werden. Eine Möglichkeit ist, das Festschreiben zu wiederholen, in der Hoffnung, dass es zu einem späteren Zeitpunkt erfolgreich sein wird. Eine befriedigende Lösung ist dies jedoch nicht.

Dieser Ablauf wird durch die Abbildung 5.9 (S. 61) verdeutlicht. Der linke Pfad stellt den fehlerfreien Ablauf dar, sobald jedoch ein Fehler auftritt, wird die Ausführung abgebrochen (rechter Pfad). Nur wenn die Transaktion erfolgreich abgeschlossen wurde, kann durch die Synchronisation eine syntaktisch identische Datei erzeugt werden (siehe Kapitel 2.1.2 (S. 8)). Bei der Synchronisation von Verzeichnissen ist nur entscheidend, dass diese zum richtigen Zeitpunkt existieren. Wird ein Verzeichnis erzeugt und Dateien in diesem abgelegt, so muss dieses Verzeichnis vor der Erzeugung der ersten Datei existieren. Eine syntaktische Gleichheit von Verzeichnissen existiert jedoch nicht.

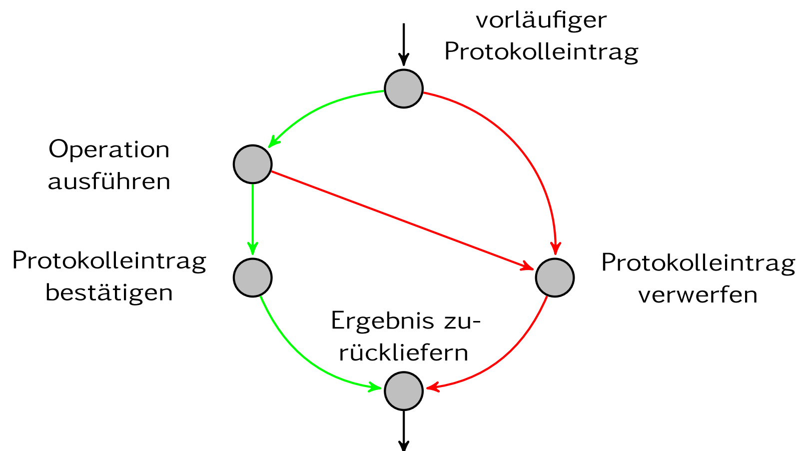

Abbildung 5.9.: Transaktionales Verhalten während der Protokollierung

Um nun eine Synchronisation zu ermöglichen, müssen die protokollierten Informationen an bekannte Synchronisationspartner übermittelt werden. Dort können diese Informationen dann, sofern keine Konflikte erkannt wurden, verwendet werden, um die jeweilige Operation erneut auszuführen und so eine Synchronisation von Dateien und Verzeichnissen zu ermöglichen. Die Kommunikation und die Erkennung von Konflikten wird zu einem späteren Zeitpunkt genauer betrachtet.

Die meisten zu protokollierenden Operationen arbeiten zustandslos. Beispielsweise kann eine Datei einfach durch den Aufruf `unlink()` unter Angabe des Pfades gelöscht werden. Vor und nach der Ausführung dieser Operation muss keine Umgebung erzeugt werden. Einige Operationen jedoch benötigen eine solche Umgebung. Die Operation `write()` zum Beispiel erfordert es, dass zuerst eine Datei zum Schreiben geöffnet wird, bevor die Operation erfolgreich ausgeführt werden kann. Abschließend muss die Datei wieder geschlossen werden. Spätestens zu diesem Zeitpunkt muss das Dateisystem die Änderungen auf den Massenspeicher schreiben (vgl. Tanenbaum 2003*b*, S. 416). Nachfolgend sind die wenigen zustandsabhängigen und für eine Synchronisation notwendigen Operationen dargestellt.

- `creat()`
- `write()`
- `ftruncate()`

Diese Operationen erzeugen und öffnen eine Datei (`creat()`) beziehungsweise erfordern eine geöffnete Datei (`write()` und `ftruncate()`), um erfolgreich zu arbeiten und sind somit nicht zustandslos. Dennoch können diese Operationen ebenfalls als zustandslose Operationen protokolliert werden. Dadurch vereinfacht sich auf der einen Seite die Protokollierung, auf der anderen Seite fallen so Möglichkeiten der Optimierung weg. Dies soll nachfolgend an einem Beispiel kurz verdeutlicht werden:

Beispiel 9 *Einige Editoren wie der* GNU Emacs[2] *oder* VIM[3] *führen folgende Operationen aus, wenn eine geöffnete Datei gespeichert wird:*

1. `ftruncate(..., 0)`
2. `write(...)`

Im ersten Schritt wird die Datei durch den Aufruf `ftruncate(..., 0)` *geleert und im zweiten Schritt werden dann die neuen Daten in die Datei geschrieben.*

Bearbeitet ein Anwender nun eine Datei und speichert diese regelmässig, so werden alle Aufrufe protokolliert. In diesem Beispiel ergibt sich daraus eine Folge bestehend aus den beiden genannte Aufrufen. Wie leicht erkennbar ist, sind nur die letzten beiden Aufrufe notwendig, um eine syntaktisch identische Datei zu erzeugen. Daher könnten prinzipiell alle vorherigen Aufrufe dieser beiden Operationen aus dem Protokoll entfernt werden. Dies setzt selbstverständlich voraus, dass immer die Operation `ftruncate(..., 0)` *direkt gefolgt von* `write()` *im Protokoll erscheint und somit dem beschriebenen Verhalten entspricht.*

Ohne Entfernen der obsoleten Operationen aus dem Proktoll werden während der Synchronisation mehr Operationen als notwendig übertragen und erneut ausgeführt. Ebenso ist das Protokoll aufgrund dieser Einträge größer als notwendig. Dennoch soll dieses Verhalten beibehalten werden, um eine möglichst einfache Protokollierung zu ermöglichen. Eine Optimierung kann zu einem späteren Zeitpunkt immer noch erfolgen, wie Sutter und Alexandrescu empfehlen:

> *Always remember:*
>
> ***It is far, far easier to make a correct program fast than it is to make a fast program correct.***
>
> *(Sutter und Alexandrescu 2005, S. 16)*

[2] Richard M. Stallman und the GNU Project. *GNU Emacs*. URL: http://www.gnu.org/software/emacs/ (besucht am 10. 09. 2008).

[3] Bram Moolenar u. a. *VIM*. URL: http://www.vim.org/ (besucht am 10. 09. 2008).

Besonders wichtig ist es, dass die Reihenfolge, in der die Operationen ausgeführt wurden, ebenfalls im Protokoll festgehalten wird. Diese Reihenfolge kann als *happens-before*-Beziehung nach Lamport angesehen werden (vgl. Tanenbaum und van Steen 2008, S. 275) und muss bei der wiederholten Ausführung der Operationen berücksichtigt werden. Werden die Operationen nicht in exakt der gleichen Reihenfolge ausgeführt, in der sie ursprünglich ausgeführt wurden, so besteht die Gefahr, dass die Pfade – hauptsächlich Dateien – nach erfolgter Synchronisation bei beiden Synchronisationspartnern unterschiedlich sind. Daher muss die Reihenfolge der Operationen, die je Pfad ausgeführt wurden, im Protokoll festgehalten werden, so dass diese Reihenfolge bei der wiederholten Ausführung der Operationen exakt wieder hergestellt werden kann. Dies kann beispielsweise einfach durch einen Zähler je Pfad erreicht werden. Dieser Zähler – oder besser diese Version – muss dann für jede neu protokollierte Operation inkrementiert werden. Sortiert man die Operationen je Pfad zu einem späteren Zeitpunkt anhand dieser Version, so erhält man die ursprüngliche Reihenfolge, in der die Operationen ausgeführt wurden.

Wie später noch genauer beschrieben wird, hilft diese Version auch, die Erkennung von Konflikten zu vereinfachen.

Abschließend soll beispielhaft eine Tabelle (siehe 5.1 (S. 63)) mit einigen protokollierten Operationen dargestellt werden.

Version	Zeitpunkt	Pfad	Operation
7	2008-09-09 00:00:01	tmp/test	write(...)
4	2008-09-09 00:00:07	tmp/image.png	chmod(...)
8	2008-09-09 00:00:08	tmp/test	chmod(...)
9	2008-09-09 00:00:10	tmp/test	chown(...)
5	2008-09-09 00:00:11	tmp/image.png	chown(...)
6	2008-09-09 00:02:07	tmp/image.png	write(...)

Tabelle 5.1.: Auszug einiger protokollierter Operationen

Unidirektionale Synchronisation

Wie bereits erwähnt, sollen die protokollierten Operationen zur Synchronisation genutzt werden. Dazu müssen sie an Synchronisationspartner, also beispielsweise andere Computer, auf denen ebenfalls diese Anwendung läuft, übermittelt werden. Zudem kann die Anwendung ebenfalls auf einem Computer mehrfach genutzt werden, um beispielsweise eine Datensicherung auf eine externe Festplatte vorzunehmen.

Dieser Anwendungsfall soll zunächst betrachtet werden, da eine Datensicherung einer unidirektionalen Synchronisation entspricht und zudem die Betrachtung vereinfacht, da keine Konflikte auftreten können.

Als erstes müssen die protokollierten Operationen, wie sie in Tabelle 5.1 (S. 63) beispielhaft dargestellt sind, an den Synchronisationspartner übertragen werden. Wie diese Kommunikation erfolgen kann, wird später betrachtet. Die Reihenfolge, in der die Operationen übertragen werden, ist unerheblich, da jede Operation eine Version enthält, die in Kombination mit dem Pfad eine Operation aus dem Protokoll eindeutig identifiziert. Nach Erhalt der Operationen müssen diese je Pfad in der richtigen Reihenfolge ausgeführt werden. Wird beispielsweise durch die Ausführung von Operationen eine neue Datei erzeugt, so muss sichergestellt werden, dass das Verzeichnis, in dem die neue Datei erzeugt werden soll, existiert.

Wenn also Operationen existieren, die das Verzeichnis erzeugen, so sind diese zuerst auszuführen. Beispielsweise könnten die Operationen zusätzlich zur Version noch anhand der Ausführungszeit sortiert werden. Sind diese Zeiten identisch, so können auch weitere Details der Operationen berücksichtigt werden. So können erst die Operationen ausgeführt werden, die auf Verzeichnissen operieren, und anschließend die Operationen, die auf Dateien operieren.

Um die zu übertragenen Daten möglichst gering zu halten, sollte lokal der Zeitpunkt der letzten erfolgreichen Synchronisation für jeden Synchronisationspartner gespeichert werden. So kann einfach entschieden werden, welche Operationen seit diesem Zeitpunkt dem Protokoll hinzugefügt wurden. Diese neuen Operationen können dann übertragen werden und jede Operation sollte zudem explizit bestätigt werden, um sicherzustellen, dass der Synchronisationspartner die Operation erfolgreich seinem Protokoll hinzugefügt hat.

Alternativ besteht die Möglichkeit, zunächst die aktuelle Version jedes Pfades des Synchronisationspartners zu erfragen und anhand dieser Information zu entscheiden, welche Operationen zu übertragen sind. Jedoch werden anfangs wieder Informationen gesammelt, übertragen und verglichen. Daher sollte die zuvor vorgestellte Möglichkeit – die Speicherung des letzten Synchronisationszeitpunkts – genutzt werden. Sollten dann wider erwarten Versionen fehlen, können diese leicht erneut übertragen werden.

Auch während einer aktiven Synchronisation kann mit den zu synchronisierenden Dateien und Verzeichnissen weiter gearbeitet werden, da dem Protokoll immer nur neue Operationen hinzugefügt werden.

Bidirektionale Synchronisation

In diesem Abschnitt soll die im vorherigen Abschnitt vorgestellte unidirektionale Synchronisation erweitert werden, so dass eine bidirektionale Synchronisation zwischen zwei Verzeichnissen erfolgen kann.

Ausgehend von der unidirektionalen Synchronisation sollen nachfolgend die notwendigen Anpassungen vorgestellt werden. Die Synchronisation erfolgt nach wie vor durch das Übertragen der einzelnen Operationen, in diesem Fall jedoch in beide Richtungen. Daher muss es nun möglich sein, Konflikte sicher zu erkennen, um Datenverluste zu vermeiden.

Wenn weiterhin mit dem Verzeichnis gearbeitet werden soll, während eine Synchronisation ausgeführt wird, darf eine Datei oder ein Verzeichnis nicht gleichzeitigt durch den Anwender und die Synchronisation verändert werden. Dadurch würde eine Inkonsistenz zwischen der Datei oder dem Verzeichnis und dem Protokoll entstehen. In so einem Fall würde die Datei oder das Verzeichnis nicht mehr den Zustand des Protokolls widerspiegeln. Da aber nur das Protokoll zur Synchronisation verwendet wird, würde die Version der Datei oder des Verzeichnisses nicht der synchronisierten Version entsprechen. Eine Datei oder ein Verzeichnis darf daher nur entweder durch den Anwender oder die Synchronisation verändert werden.

Da sich die bidirektionale Synchronisation in erster Linie durch das mögliche Auftreten von Konflikten von der unidirektionalen Synchronisation unterscheidet, soll im nächsten Abschnitt die Konflikterkennung und anschließend die Konfliktlösung vorgestellt werden.

5.4.2. Konflikterkennung

Zur sicheren Konflikterkennung reichen die bisherigen Informationen des Protokolls über die Operationen noch nicht aus. Bei einer bidirektionalen Synchronisation kann der Fall eintreten, dass eine Version durch die gleiche Operation bei zwei Synchronisationspartnern erzeugt werden kann. Mit den bisherigen Informationen über die Operationen lassen sich diese beiden Einträge im Protokoll nicht unterscheiden. Auch wenn der Zeitpunkt der Operation unterschiedlich sein sollte, so ist diese Information – wie mehrfach angesprochen (siehe Kapitel 2.1.1 (S. 5) und Tanenbaum und van Steen (2008, S. 300)) – sehr unzuverlässig und soll daher nicht weiter berücksichtigt werden.

Es muss daher eine andere Möglichkeit genutzt werden, jede Operation eindeutig identifizieren zu können. Dazu bieten sich beispielsweise *UUIDs* (Microsoft, Refactored

Networks, LLC und DataPower Technology, Inc. 2005) an, da diese speziell zu diesem Zweck entwickelt worden sind, zur dezentralen Erzeugung eindeutiger Kennungen. Wird das Protokoll um diese Information erweitert und für jeden neu erzeugten Eintrag – also jede neue Operation – eine *UUID* erzeugt und ebenfalls abgelegt, so können später die *UUIDs* zweier Operationen zusätzlich zum Vergleich verwendet werden.

Wird nun während einer Synchronisation festgestellt, dass zwei Operationen existieren, bei denen die Version und der Pfad identisch sind, kann durch Vergleich der *UUID* festgestellt werden, ob eine Operation während einer Synchronisation als Kopie der anderen Operation entstanden ist. In diesem Fall wären beide *UUIDs* identisch.

Dieser Vergleich vereinfacht somit die Erkennung von Konflikten, ist jedoch auch sehr eng gefasst. Formal muss folgende Bedingung erfüllt sein:

Definition 8 *Sei v_1, v_2 die Versionsnummer der beiden Pfade, p_1, p_2 die jeweiligen Pfade und id_1, id_2 die jeweilige eindeutige Kennung, dann muss zur Erkennung eines Konflikts gelten:*

$$\begin{aligned} v_1 = v_2 \quad &\wedge \\ p_1 = p_2 \quad &\wedge \\ id_1 \neq id_2 \end{aligned}$$

Durch diese sehr eng gefasste Definition werden jedoch auch Konflikte erkannt, die tatsächlich keine sind. Dies soll an einem einfachen Beispiel verdeutlicht werden.

Beispiel 10 *Angenommen eine Datei* `test` *wurde ohne Konflikte synchronisiert. Nun wird bei beiden Synchronisationspartnern unabhängig voneinander eine Kopie dieser Datei mit gleichem Namen angefertigt. Wenn anschließend eine Synchronisation erfolgt, wird nach der Definition 8 (S. 66) ein Konflikt erkannt, obwohl nach der Definition 4 (S. 9) beide Dateien syntaktisch identisch sind.*

In diesem Beispiel wird also ein Konflikt erkannt, obwohl keiner vorliegt.

Um diese fehlerhafte Erkennung von Konflikten zu beheben, müssten auch die jeweiligen Operationen inklusive aller Parameter miteinander verglichen werden. Dieser Vergleich ist in jedem Fall komplexer als ein einfacher Vergleich zweier *UUIDs* und soll als mögliche Optimierung vorgemerkt werden, auch wenn ohne diese Verbesserung die Gefahr besteht, fehlerhaft Konflikte zu erkennen.

Version	Pfad	ID	SP		Konflikt		Version	Pfad	ID	SP
3	/test	a7	1	←	kein Konflikt	→	3	/test	a7	2
4	/test	b8	1	←	**Konflikt**	→	4	/test	b3	2
5	/test	f7	1	←	**Konflikt**	→	5	/test	c6	2

Tabelle 5.2.: Konflikterkennung

Die Tabelle 5.2 (S. 67) verdeutlicht die Erkennung eines Konflikts. Die Spalte *ID* enthält eine angedeutete *UUID*[4] und die Spalte *SP* die Kennung des jeweiligen Synchronisationspartners. Die Version 3 ist bei beiden Synchronisationspartnern identisch, danach haben sich die beiden Dateien unterschiedlich weiterentwickelt.

Die Darstellung in Abbildung 5.10 (S. 67) zeigt die unterschiedliche Entwicklung als gerichteten Graph, so dass die parallele Entwicklung beider Dateien deutlich sichtbar ist. Der Knoten enthält zur Information noch die Version der Operation und – durch zwei hexadezimale Ziffern angedeutet – die *UUID*.

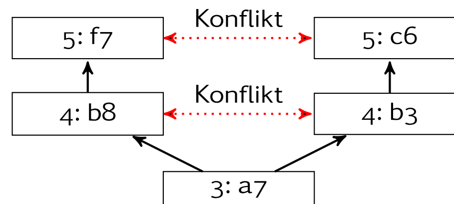

Abbildung 5.10.: Beide Zweige der Datei `/test` (siehe Tabelle 5.2 (S. 67))

5.4.3. Konfliktlösung

Im vorherigen Abschnitt wurde die Konflikterkennung (Kapitel 5.4.2 (S. 65)) beschrieben. In diesem Abschnitt wird nun der Fokus auf die Lösung eines erkannten Konflikts gerichtet.

Wenn ein Konflikt erkannt wurde, bedeutet dies, dass es zwei Entwicklungszweige dieser Datei oder dieses Verzeichnisses gibt. Es sind nur zwei Entwicklungszweige, da momentan nur zwei Synchronisationspartner betrachtet werden. Tatsächlich hängt die Anzahl der möglichen Zweige von der Anzahl der Synchronisationspartner ab. Dies wurde bereits in Abbildung 5.10 (S. 67) deutlich dargestellt. Zudem können die beiden Zweige unterschiedlich viele protokollierte Operationen enthalten. In diesem Beispiel werden nur zwei Operationen gezeigt, um die Übersichtlichkeit zu wahren.

[4] Eine *UUID* besteht als Zeichenkette aus 36 Zeichen und wäre somit für die Darstellung viel zu breit.

Ziel der Konfliktlösung ist es, diese beiden Zweige wieder in einen Zweig zu überführen. Da dies durch den Anwender erfolgen muss, sollten dem Anwender beide in Konflikt stehenden Versionen angeboten werden, so dass ihm alle Informationen für eine Konflitklösung zur Verfügung stehen. Die Lösung des Konflikts ist dann auf unterschiedliche Arten möglich, die nachfolgend vorgestellt werden sollen.

Eine mögliche Konfliktlösung verwendet eine neue Operation, die die beiden neuesten Versionen beider Zweige in eine neue Version zusammenführt. Dies ist beispielsweise die Variante, die *Mercurial* (siehe Kapitel 3.5 (S. 30)) verwendet, um zwei Zweige zusammenzuführen. Ein Vorteil ist, dass weiterhin alle anderen Versionen erhalten bleiben, was für ein Versionskontrollsystem wie *Mercurial* notwendig ist. Eine entsprechende Lösung ist in Abbildung 5.11 (S. 68) dargestellt.

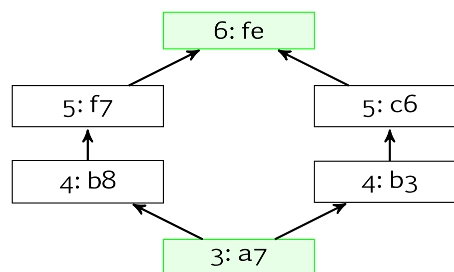

Abbildung 5.11.: Zusammenführung beider Zweige in eine neue Version

Eine weitere Möglichkeit der Konfliktlösung ist, die Operationen eines Zweiges zu entfernen. Es ist dann nicht notwendig, eine neue Operation zu erzeugen, die beide Zweige zusammenführt. Aus Sicht eines Versionskontrollsystems wie *Mercurial* (siehe Kapitel 3.5 (S. 30)) ist dieses Vorgehen nicht akzeptabel, da einmal erzeugte Versionen und somit Teile der Versionshistorie gelöscht werden. Für eine Synchronisation ist dies jedoch eine ebenfalls denkbare Lösung, da die Operationen nur zum Zweck der Synchronisation protokolliert werden und nicht, um ein Versionskontrollsystem zu implementieren. Diese Möglichkeit ist in Abbildung 5.12 (S. 69) dargestellt. Der entfernte linke Zweig ist entsprechend mit gestrichelten Linien dargestellt.

5.4.4. Optimierungsmöglichkeiten

Nachdem der Kern des synchronisierenden Dateisystems vorgestellt wurde, sollen nun einige wenige Optimierungsmöglichkeiten genannt werden. Diese können beispielsweise die Synchronisation beschleunigen oder den für die Protokollierung der Operationen genutzten Speicherplatz reduzieren.

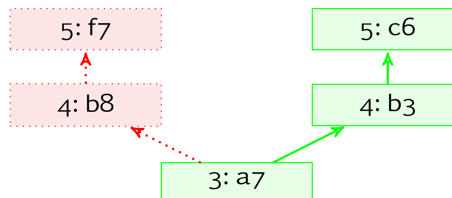

Abbildung 5.12.: Entfernung der Operationen des linken Zweiges

Um Speicherplatz zu sparen, besteht die Möglichkeit, nicht mehr benötigte Operationen zu löschen. Nicht mehr benötigte Operationen sind solche, deren Zeitpunkt der Ausführung älter ist als der älteste Zeitpunkt der letzten Synchronisation. Dies sind alle Operationen, die zum Zeitpunkt des Entfernens bereits an alle Synchronisationspartner übermittelt wurden. Kommen neue Synchronisationspartner hinzu, so müssen die gesamten Dateien inklusive Berechtigungen übertragen werden. Anschließend ist nur noch die Synchronisation mit Hilfe der Operationen notwendig. Das Übertragen einer vollständigen Datei kann in Einzelfällen auch dazu führen, dass – im Vergleich zum Übertragen aller Operationen – weniger Daten zu übertragen sind. Dies könnte insbesondere dann der Fall sein, wenn, wie in Beispiel 9 (S. 62) beschrieben, teilweise unnötige Operationen protokolliert und übertragen werden.

Es existiert eine weitere, sehr einfache Möglichkeit, den Speicherplatzbedarf und die Dauer der Synchronisation zu verringern. Dateien oder Verzeichnisse, die zwischen zwei Synchronisationen angelegt und wieder gelöscht wurden, brauchen nicht synchronisiert werden. Das heißt, die zugehörigen, protokollierten Operationen können wieder aus dem Protokoll entfernt werden. Zu solchen Dateien gehören in der Regel temporäre Dateien, die von Anwendungen während der Bearbeitung angelegt werden. Beispielsweise legen verschiedene Editoren oder Office Pakete Sicherungskopien geöffneter und modifizierter Dateien an.

Ein etwas anderer Ansatz, der aber dem gleichen Zweck dient, ist, gewisse Dateien von der Synchronisation auszuschließen. Mit Hilfe regulärer Ausdrücke oder anderer Muster können Datei- oder Verzeichnisnamen definiert werden, deren Operationen nicht protokolliert werden sollen.

Die Konflikterkennung lässt sich ebenfalls etwas verbessern, so dass mehr Konflikte automatisch gelöst werden können. Die Definition 8 (S. 66) zur Erkennung von Konflikten ist sehr eng gefasst, da sie beispielsweise keine kommutativen Operationen (siehe Kapitel 2.1.2 (S. 11)) zulässt. Ebenso wird nicht erkannt, wenn bei beiden Dateisystemen die gleichen Änderungen ohne zwischenzeitliche Synchronisation durchgeführt

wurden, da nicht die Operationen direkt verglichen werden, sondern nur ihre jeweilige eindeutige Kennung. Es stellt sich an dieser Stelle die Frage, wie häufig diese Fälle in der Praxis auftreten, was jedoch in erster Linie von den Nutzungsgewohnheiten des Anwenders abhängt. Je häufiger diese Fälle auftreten, desto sinnvoller ist es, die Operationen und alle Parameter zu vergleichen. Je nach Operation wird dieser Vergleich unterschiedlich aufwändig sein, da auch alle Parameter zu vergleichen sind. Nur wenn die Parameter ebenfalls identisch sind, wurde eine identische Operation ausgeführt. In diesem Fall wäre dann kein Konflikt erkannt worden, auch wenn die eindeutigen Kennungen unterschiedlich sind. Ein Synchronisationspartner müsste dann sein Protokoll aktualisieren und die entsprechende Operation mit der anderen Operation überschreiben beziehungsweise die Kennung der Operation aktualisieren.

5.5. Kommunikation

Die Kommunikation, die zwischen zwei Synchronisationspartnern erfolgen muss, damit eine Synchronisation stattfinden kann, wurde bisher nur erwähnt, aber nicht näher betrachtet. Bevor jedoch die Kommunikation in Bezug auf das synchronisierende Peer-to-Peer Dateisystem betrachtet wird, sollen wichtige Anforderungen genannt werden.

Bisher wurde auf das Thema Sicherheit nicht eingegangen. Im Rahmen der Synchronisation war dies nicht weiter notwendig, weil das Ziel ist, dass ein Anwender seine eigenen Dateien zwischen seinen Computern synchronisiert. Ziel ist es nicht, dass ein Anwender sich auf einfache Art und Weise eine Kopie der Dateien eines anderen Anwenders anfertigt und diese automatisch aktualisiert werden. Dies ist auch ein interessantes Szenario, aber nicht Gegenstand dieses Konzepts. Wenn jedoch eine Synchronisation zwischen zwei Computern über ein Netzwerk stattfindet, so muss sichergestellt werden, dass nur der Besitzer Zugriff auf die persönlichen Dateien erhält, auch wenn es sich um das lokale Netzwerk des Anwenders handelt. Erfolgt die Kommunikation über ein unsicheres Netzwerk – wie etwas das Internet –, dann muss besonderer Wert auf die Sicherheit gelegt werden (vgl. Schmeh 2007, S. 14 ff.; Tanenbaum 2003*b*, S. 624 ff.; Tanenbaum 2003*a*, S. 779 ff.; Tanenbaum und van Steen 2008, S. 413 ff.). Dazu sind prinzipiell zwei Varianten denkbar:

1. Die Authentifizierung wird an eine externe Anwendung delegiert.
2. Im Rahmen dieses Konzepts wird eine Authentifizierungsmöglichkeit entwickelt.

Da die Entwicklung einer sicheren Authentifizierung, die über eine verschlüsselte Verbindung erfolgt, nicht trivial ist, soll hier eine vorhandene Anwendung verwendet werden. Eine entsprechend ausgereifte und weit verbreitete Anwendung ist Open*SSH* (*Open*SSH)[5], die sehr flexibel eingesetzt werden kann. Auf der Open*SSH* Website ist dazu zu lesen:

> [...] Additionally, OpenSSH provides secure tunneling capabilities and several authentication methods, and supports all SSH protocol versions. (*Open*SSH, Website)

Es ist also möglich, die TCP/IP Kommunikation (vgl. Tanenbaum 2003*a*, S. 580 ff.) vorhandener Anwendungen über einen sicheren Tunnel zu leiten. Damit verhält sich Open*SSH* einem *VPN (**V**irtual **P**rivate **N**etwork)* (vgl. Schmeh 2007, S. 569) sehr ähnlich.

Eine sichere und verschlüsselte Kommunikation muss allerdings nicht erfolgen, wenn die Synchronisation beispielsweise mit einem externen Massenspeicher erfolgt. In diesem Fall wird nur eine lokale Verbindung genutzt.

Weiterhin sollten so wenig Daten wie möglich übertragen werden, um die Dauer der Synchronisation möglichst kurz zu halten. Insbesondere wenn die Kommunikation über Verbindungen mit wenig Bandbreite – wie beispielsweise Funknetze – erfolgen, verhindert dies nicht die Verwendung. Auch instabile Funkverbindungen könnten so für eine Synchronisation genutzt werden. Zudem möchten die meisten Anwender sicherlich nicht lange auf die Beendigung der Synchronisation warten.

Wenn die Synchronisation über instabile Verbindungen erfolgt, die immer wieder abbrechen, ist es notwendig, dass eine Synchronisation entsprechend reagieren kann. Wenn jedesmal die Synchronisation alle schon gesendenten Information erneut senden muss, um sicher zu funktionieren, kann in solchen Umgebung eventuell der ungünstigste Fall eintreten, dass die Synchronisation nie vollständig erfolgen kann. Daher sollte eine Synchronisation möglichst an dem Punkt erneut starten, an dem die Verbindung abgebrochen ist.

Im nächsten Abschnitt sollen diese Anforderungen in Bezug auf das vorgestellte synchronisierende Dateisystem betrachtet werden.

5.5.1. Kommunikation des synchronisierenden Dateisystems

Zunächst soll die Kommunikation zwischen zwei Synchronisationspartnern näher betrachtet werden. Anschließend soll die Kommunikation daraufhin untersucht werden, inwieweit sie die im vorherigen Abschnitt aufgestellten Anforderdungen erfüllen kann.

[5] siehe auch http://www.openssh.org/usage/graphs.html

Bisher sind drei verschiedene Nachrichten im Rahmen dieses Konzepts erwähnt worden. Die größtenteils zu übermittelnde Nachricht enthält die protokollierte Operation, die somit die eigentliche Synchronisation der Dateien und Verzeichnisse ermöglicht. Dazu zählt als eigene Nachricht selbstverständlich die Bestätigung (negativ oder positiv). Eine weitere Nachricht ist für die Lösung eines nicht automatisch lösbaren Konflikts notwendig. Je nach verwendeter Konfliktlösungsstrategie sind diese Nachrichten unterschiedlich. Es sollte jedoch nur eine Strategie der Konfliktlösung verwendet werden.

Im Rahmen der Konfliktlösung und möglicher Optimierungen wurde noch erwähnt, dass vollständige Dateien zu übermitteln seien. Einerseits könnte dazu eine weitere Nachricht definiert werden, andererseits könnte dies über gekennzeichnete Operationen simuliert werden. Werden diese in den genannten Fällen verwendet, können diese sogar ins Protokoll eingetragen werden.

Es sind jedoch noch einige weitere Nachrichten notwendig, damit die Synchronisation erfolgreich funktionieren kann. Zunächst muss die Synchronisation gestartet werden, indem die Kennungen der beiden Synchronisationspartner ausgetauscht werden. Da jedes Dateisystem mit mehreren anderen Dateisystemen Dateien synchronisieren kann, ist dies notwendig, um die empfangenen Operationen und Konflikte zuordnen zu können. Da der Anwender die Synchronisationspartner einander zuordnen muss, bevor eine Synchronisation stattfinden kann, ermöglicht dieser Austausch der Kennungen, dass die Zuordnung nur bei einem Synchronisationspartner erfolgen muss. Wenn beispielsweise der Arbeitsplatzrechner mit dem Notebook synchronisiert werden soll, ist es ausreichend, entweder dem Notebook oder dem Arbeitsplatzrechner den jeweils anderen Computer zuzuordnen.

Außerdem sollten anfangs Informationen über die unterstützen Versionen der Nachrichtenformate ausgetauscht werden. Ebenso könnte weitere Funktionalität an dieser Stelle ausgehandelt werden, wie beispielsweise das komprimierte Übertragen der Nachrichten.

Ebenso kann vor dem Beenden der Verbindung eine Nachricht ausgetauscht werden, dass vorerst nicht weiter synchronisiert werden soll. Dies ist beispielsweise hilfreich, wenn der Anwender die Möglichkeit haben soll, die Synchronisation zu beenden, etwa weil die Computer abgeschaltet werden müssen.

Um die Kommunikation über eine sichere Verbindung, die von Open*SSH* bereitgestellt wird, zu tunneln, muss die Kommunikation lediglich über TCP/IP erfolgen (vgl. Tanenbaum 2003*a*, S. 580 ff.). Die Verwendung eines bytestrom orientierten Protokolls

ist für diese Kommunikation ohnehin sinnvoll, da sichergestellt ist, dass die Daten in richtiger Reihenfolge beim Empfänger ankommen (vgl. Tanenbaum 2003*a*, S. 573 ff.). Die Verwendung ist dem Zugriff auf eine Datei ähnlich (vgl. Tanenbaum 2003*a*, S. 582), Daten können jedoch nur sequentiell geschrieben und empfangen werden.

Da eine Synchronisation zwischen den Synchronisationspartnern durch eine einzige Nachricht initiiert wird und anschließend nur die Operationen und Bestätigungen für diese ausgetauscht werden, sind wenig Zustandsinformationen von den Synchronisationspartnern zu speichern. Bei einem Verbindungsabbruch ist es daher nicht notwendig, alle schon gesendeten Daten erneut zu übertragen. Nachdem die Verbindung wiederhergestellt wurde, kann die Synchronisation erneut initiiert und direkt die nächsten Operationen übertragen werden. Allerdings kann der Fall eintreten, dass eine Operation nur teilweise empfangen wurde. Ebenso ist es möglich, dass eine Operation empfangen wurde, die Bestätigung jedoch nicht mehr gesendet werden konnte. Im ersten Fall muss eine negative Bestätigung gesendet werden, so dass die Operation erneut gesendet wird. Im zweiten Fall, muss der Sender wissen, dass eine Bestätigung fehlt und auf diese warten, darf aber die Operation nicht erneut senden.

Konflikte werden von beiden Synchronisationspartnern selbstständig erkannt, so dass keine weiteren Nachrichten hierfür zu senden sind. Lediglich die Lösung eines Konflikts muss dem Synchronisationspartner mitgeteilt werden.

5.6. Zusammenfassung

In diesem Kapitel wurde ein Konzept vorgestellt, dass eine Synchronisation von Dateien und Verzeichnissen in die Dateisystemschicht verlagert, um möglichst automatisch eine Synchronisation durchzuführen.

Dabei wurde gezeigt, dass eine Synchronisation inklusive Konflikterkennung möglich ist. In einigen Fällen können Konflikte sogar automatisch – ohne Eingriff des Benutzers – gelöst werden. Weiterhin kann ein aufwändiges Durchsuchen der zu synchronisierenden Dateien und Verzeichnisse entfallen, da alle notwendigen Informationen bereits während der Nutzung gesammelt werden. Allerdings erfordert die Speicherung der für die Synchronisation notwendigen Daten mehr Speicherplatz, als wenn etwa nur der Zeitpunkt der letzten Änderung und der kryptografische Hash-Wert jeder Datei abgelegt wird.

Um jedoch nicht noch die gesamte Verwaltung der Dateien auf dem Massenspeicher zu entwickeln, soll diese Aufgabe an ein existierendes Dateisystem delegiert werden.

Dies ermöglicht den Einsatz vorhandener, stabiler Dateisysteme und der Fokus kann auf die Synchronisation gerichtet werden. Somit handelt es sich bei dem beschriebenen synchronisierenden Peer-to-Peer Dateisystem viel mehr um eine Schicht oberhalb eines existierenden Dateisystems als um ein vollkommen eigenständiges Dateisystem. Aus Sicht der Anwendungen und des Anwenders ist dies jedoch nicht ersichtlich, da diese mit dem synchronisierenden Peer-to-Peer Dateisystem in der gleichen Art und Weise arbeiten wie mit anderen Dateisystemen auch.

6. Implementierung

In diesem Kapitel sollen die Komponenten der prototypischen Implementierung des in Kapitel 5 (S. 47) entwickelten synchronisierenden Peer-to-Peer Dateisystems vorgestellt werden.

Die Entwicklung soll in der Programmiersprache C++ erfolgen, da so objektorientiert, effizient und mit hoher Abstraktion entwickelt werden kann (vgl. Meyers 2006, S. 17 ff.; Alexandrescu 2003, S. 15; Sutter und Alexandrescu 2005, S. 3). Außerdem lässt sich so die zentrale in der Programmiersprache C entwickelte Bibliothek *FUSE* (vgl. FUSE) sehr einfach einbinden.

Bis auf die Bibliothek *Boost*, die in C++ entwickelt wird, werden alle anderen verwendeten Bibliotheken in der Programmiersprache C entwickelt, so dass hier ebenfalls die Integration vereinfacht wird. Weiterhin lassen sich in der Programmiersprache C++ viele Anwendungsfälle einfach umsetzen, indem beispielsweise Muster wie das *Scoped Locking* (vgl. Schmidt u. a. 2004, S. 325 ff.) verwendet werden. Außerdem sind Richtlinien, wie *single entry, single exit* (vgl. Sutter und Alexandrescu 2005, S. 3, S. 24 f.) nicht notwendig, um übersichtlichen, sicheren Quelltext zu schreiben, der die definierten Invarianten sowie Vor- und Nachbedingungen erfüllt.

Bevor die einzelnen Komponenten, wie sie in Abbildung 5.8 (S. 58) veranschaulicht wurden, betrachtet werden, sollen zunächst zentrale und in allen Komponenten verwendete Bibliotheken untersucht und vorgestellt werden.

6.1. Integration externer Bibliotheken

Bei der Vorstellung der Konzeption des synchronisierenden Peer-to-Peer Dateisystems (Kapitel 5 (S. 47)), wurden viele notwendige Komponenten genannt. So müssen Operationen eindeutig identifizierbar, protokollierbar und leicht anhand eines Zeitraums für eine Synchronisation selektierbar sein. Sie müssen zudem über eine Kommunikationsleitung an andere Synchronisationspartner übermittelt werden können. Dies sind viele

Anforderungen, die mit Hilfe vorhandener Bibliotheken einfach und auch effizient erfüllt werden können.

Daher werden zunächst verschiedene Software Bibliotheken hinsichtlich ihrer Funktionalität und ihrer Integration in die Implementierung untersucht.

Das von Schmidt und Huston in C++ entwickelte *Adaptive Communication Environment (ACE)*[1], das in *C++ Network Programming – Volume 1* (Schmidt und Huston 2005) und *C++ Network Programming – Volume 2* (Schmidt und Huston 2006) ausführlich beschrieben wird, vereinfacht an vielen Stellen die Implementierung. Es stellt eine Fassade (vgl. Gamma u. a. 1996, S. 212) für viele C Systemfunktionen zur Verfügung. Die Bibliothek stellt nicht nur Funktionen zur Kommunikation zur Verfügung, sondern es werden auch Threads und Prozesse vom Betriebssystem abstrahiert. Es zeigt sich jedoch an vielen Stellen deutlich die Nähe zu den C Systemfunktionen. Außerdem wird versucht, kompatibel zu älteren C++ Compilern zu sein, und somit werden kaum aktuelle Techniken der C++ Entwicklung verwendet. Daher soll das *Adaptive Communication Environment* hier nicht verwendet werden.

Die von *Trolltech*[2] entwickelte Bibliothek *QT* stellt in der Version 4.4 eine sehr umfangreiche Implementierung dar, setzt aber – ursprünglich wegen fehlender Möglichkeiten der C++ Compiler – auf externe Hilfsprogramme, um ein effizientes Nachrichtensystem zu implementieren. Einige Komponenten existieren schon in *QT*, wie beispielsweise eine Klasse zur Erzeugung und Nutzung von *UUIDs* oder Anbindung an Datenbanken. Zudem ist das integrierte Nachrichtensystem zentraler Bestandteil der Bibliothek, ohne das die Bibliothek nicht sinnvoll genutzt werden kann. Da *QT* sehr umfangreich ist, sollte zunächst *QT* auf eine Implementierung der gewünschten Funktionalität untersucht werden. Werden externe Bibliotheken eingebunden, so müssen diese noch mit Hilfe einer Fassade (vgl. Gamma u. a. 1996, S. 212) in das Nachrichtensystem integriert werden. Dies erschwert die Integration externer Bibliotheken, zumal so auch Funktionalität mehrfach vorhanden sein kann. Aus den genannten Gründen soll *QT* ebenfalls nicht verwendet werden.

Im Folgenden soll *Boost*[3] untersucht werden. Dieses sehr umfangreiche Paket verwendet viele aktuelle C++ Techniken und stellt sehr viele verschiedene Module – in *Boost* als Bibliotheken bezeichnet – zur Verfügung. Es existiert eine Abstraktionsschicht, die viele Eigenheiten verschiedener C++ Compiler berücksichtigt, so dass *Boost* mit sehr vielen Compilern verwendet werden kann. Weiterhin kann der Entwickler nur

[1] siehe `http://www.riverace.com/`
[2] siehe `http://www.trolltech.com/`
[3] siehe `http://www.boost.org/`

die Module (Bibliotheken) verwenden, die er benötigt. Die Verwendung von *Boost* mit anderen externen Bibliotheken ist problemlos möglich, da die einzelnen Bibliotheken von *Boost* gut gekapselt sind.

Einige Bibliotheken von *Boost* werden sogar sehr wahrscheinlich in die nächste Generation der *C++ Standard Template Library (STL)* einfließen beziehungsweise sind bereits als *Technical Report* – einer Art Testbereich für neue Funktionalität – verfügbar.[4] Weiterhin halten Sutter und Alexandrescu zu *Boost* fest:

> [...] one of the most highly regarded and expertly designed C++ library projects in the world. (Sutter und Alexandrescu 2005, S. 147)

Meyers formuliert sogar in *Tipp 55: Machen Sie sich mit Boost vertraut*:

> Suchen Sie eine Sammlung qualitativ hochwertiger, plattform- und compilerunabhängiger Open-Source-Bibliotheken? Wenden Sie sich an Boost. [...] Möchten Sie einen Blick in die Zukunft von C++ wagen? Wenden Sie sich an Boost. (Meyers 2006, S. 308)

Aufgrund der hohen Qualität, der flexiblen Integration, auch mit anderen Bibliotheken, und den umfangreichen und generischen Implementierungen soll *Boost* in dieser prototypischen Implementierung des synchronisierenden Dateisystems verwendet werden.

Von *Boost* werden verschiedene Bibliotheken genutzt, die viele kleinere Anforderungen vereinfachen. Dazu gehören verschiedene Algorithmen, um Zeichenketten zu durchsuchen und zu manipulieren, und eine Hilfsklasse, die das Kopieren eines Objekts verhindert. In *Boost* existiert mit *boost::shared_ptr* eine Implementierung, die eine *RAII (Resource Acquisition Is Initialization)* Strategie für Zeiger zur Verfügung stellt (vgl. Alexandrescu 2003, S. 24 f.; Meyers 2006, S. 81 ff.). Diese ermöglicht es, die Verwaltung des Zeigers an die Implementierung des *boost::shared_ptr* zu delegieren, so dass der Speicher, auf den der Zeiger verweist, freigegeben wird, wenn kein *boost::shared_ptr* Objekt mehr diesen Zeiger verwendet. Dazu verwendet die Implementierung einen Zähler, der der Anzahl der Objekte entspricht, die diesen Zeiger referenzieren. Wenn dieser Zähler den Wert 0 annimmt, wird der Speicher freigegeben. Damit können an vielen Stellen Zeiger sicherer genutzt werden, da auch innerhalb der Implementierung ein Zugriff auf einen ungültigen Zeiger einen Fehler meldet. Zudem stellt die Implementierung sicher, dass keine Resourcenlecks – in diesem Fall Speicherlecks – auftreten.

[4] siehe auch `http://www.boost.org/`

Des Weiteren enthält *Boost* eine Bibliothek, die plattformunabhängig flexible Kommunikationsmöglichkeiten enthält. So ist eine synchrone als auch asynchrone Kommunikation einfach umsetzbar und kann über verschiedene Protokolle wie TCP, UDP oder UNIX-Domain Sockets erfolgen. Durch diese Bibliothek wird die Implementierung der Netzwerkkommunikation stark vereinfacht.

Außerdem existieren mit *boost::function* und *boost::bind* zwei Bibliotheken, die es ermöglichen, beliebige Funktionsaufrufe zu kapseln, Parameter zu binden und zu einem späteren Zeitpunkt auszuführen. Diese beiden Bibliotheken ermöglichen eine einfache Implementierung eines *Scope Guards*, der definierte Aktionen beim Verlassen des Gültigkeitsbereiches bei Bedarf ausführt. Nachfolgend soll die Implementierung und auch die Anwendung erläutert werden.

Der in dieser Arbeit vorgestellte *Scope Guard* ist eine vereinfachte Form der von Alexandrescu und Marginean veröffentlichten und in der *Loki Bibliothek* [5] enthaltenen Implementierung (vgl. Alexandrescu und Marginean 2000). Das *Scope Guard* Muster verwendet die gleiche Idee wie das *Scoped Locking* Muster (vgl. Schmidt u. a. 2004, S. 325 ff.). Ein Objekt wird verwendet, um beim Verlassen des Gültigkeitsbereichs eine Aktion auszuführen. Das *Scoped Locking* Muster aktiviert eine Sperre im Konstruktor, also bei Erzeugung des Objekts, und löst diese Sperre wieder im Destruktor. Der Vorteil ist, dass das Lösen der Sperre implizit ausgeführt wird, unabhängig davon, wie die Funktion verlassen wird. Dies vereinfacht die Entwicklung und der Quelltext bleibt übersichtlich. Im Gegensatz zum *Scoped Locking* wird jedoch beim *Scope Guard* keine Resource im Konstruktor belegt, stattdessen enthält die Implementierung eine zusätzliche Funktion, um das Ausführen der Aktion im Destruktor zu unterbinden.

Das Listing 6.1 (S. 79) zeigt beispielhaft einen möglichen Einsatz. Sollte der Funktionsaufruf in Zeile 15 nicht erfolgreich sein und `false` zurückliefern, so wird die Funktion `some_func()` verlassen. Der *Scope Guard* stellt dann sicher, dass die Klassenfunktion `file::unlink()` aufgerufen und die Datei gelöscht wird. Wird die Funktion `long_running_function()` erfolgreich ausgeführt, wird vor dem Verlassen durch den Aufruf `file_guard.commit()` in Zeile 23 die Ausführung von `file::unlink()` unterdrückt, so dass die Datei nicht gelöscht wird.

Da *Boost* noch keine Unterstützung für die Verwendung von *UUIDs* bietet, muss eine weitere separate Bibliothek genutzt werden, die diese Funktionalität zur Verfügung stellt. Momentan wird die Implementierung einer Bibliothek der Entwickler des *ext3* Dateisystems verwendet (vgl. ext2/ext3 Entwickler). Es mag als Nachteil erscheinen,

[5] siehe `http://loki-lib.sourceforge.net/`

```
bool some_func() {
  std::string filename("/tmp/temp");
  /** Datei erzeugen und öffnen. */
  std::fstream file(filename.c_str(), std::ios::trunc |
                                      std::ios::in |
                                      std::ios::out);
  if(!file) {
    /** Datei konnte nicht angelegt werden, Fehler melden. */
    return false;
  }

  /** Datei im Fehlerfall löschen. */
  util::scope_guard file_guard(boost::bind(&file::unlink,
                                           filename));
  if(!long_running_function(file)) {
    /** Fehlerfall, Datei wird gelöscht. */
    return false;
  }
  file.close();

  /** Ausführung erfolgreich, Datei muss erhalten bleiben,
   *  Ausführung der gespeicherten Operation unterbinden. */
  file_guard.commit();

  /** Erfolg melden. */
  return true;
}
```

Listing 6.1: Beispiel für den Einsatz des *Scope Guards*

wenn viele verschiedene Bibliotheken verwendet werden. Durch die Verwendung einer Fassade (vgl. Gamma u. a. 1996, S. 212 ff.) ergibt sich jedoch die Flexibiltät, zu einem späteren Zeitpunkt einzelne Bibliotheken auszutauschen, ohne dass diese Veränderung weitreichende Anpassungen in der gesamten Implementierung nach sich zieht. Zudem besteht so auch die Mögilchkeit, je nach Betriebssystem eine andere Bibliothek zu verwendet. Beispielsweise enthält die C Standardbibliothek von *FreeBSD* bereits Unterstützung für *UUIDs*, so dass auf diesem System eine andere Implementierung der Fassade verwendet werden könnte.

Zudem werden noch weitere externe Bibliotheken verwendet, die aber bei Betrachtung der Implementierung der jeweiligen Komponente vorgestellt werden sollen, da diese Bibliotheken nur in der jeweiligen Komponente verwendet werden. Diese werden entsprechend der *UUID* Integration ebenfalls mit Hilfe einer Fassade integriert, so dass im Falle eines Austausches die Anpassungen möglichst gering gehalten werden.

Die bisher vorgestellten Bibliotheken hingegen werden von allen Komponenten der Implementierung verwendet.

6.2. Komponenten

In diesem Abschnitt sollen Details der Implementierung einiger der in der Konzeption vorgestellten Komponenten (siehe Abbildung 5.8 (S. 58), Kapitel 5.4 (S. 56)) näher betrachtet werden. Nicht alle Komponenten sind in der prototypischen Implementierung umgesetzt worden.

Die Komponente *Anwender*, die die Schnittstelle zum Anwender darstellt, wurde nicht implementiert. Sie ermöglicht dem Anwender das Lösen von Konflikten, das Verwalten der Synchronisationspartner und die Konfiguration des synchronisierenden Dateisystems. Diese Aufgaben können in der prototypischen Implementierung ebenfalls vorgenommen werden, jedoch nicht über eine komfortabel bedienbare Schnittstelle. Zudem können noch keine Konflikte erkannt, angezeigt und gelöst werden. Daher wurde diese Komponente nicht implementiert.

6.2.1. Dateisystem Implementierung

In diesem Abschnitt soll die Dateisystem-Komponente der Implementierung betrachtet werden. Diese ist die zentrale Komponente, da sie einerseits die Änderungen an das tatsächliche Dateisystem weitergeben muss, damit die Dateien und Verzeichnisse die Änderungen auch widerspiegeln, andererseits unterstützt diese Komponente

auch die Protokollierung, da sie alle dafür notwendigen Informationen besitzt. Ohne die Protokollierung wäre auch die Synchronisation mit Hilfe der Operationen nicht möglich.

Wie schon in der Konzeption erläutert (siehe Kapitel 5.4 (S. 56)), soll das in dieser Arbeit entwickelte Dateisystem die Verwaltung der Dateien und Verzeichnisse nicht implementieren. Vielmehr schiebt sich die Implementierung zwischen die Anwendung und ein vorhandenes Dateisystem. Um die Implementierung zu vereinfachen und um unabhängiger vom Betriebsystem zu sein, soll eine externe Bibliothek verwendet werden, die die gewünschte Implementierung ermöglicht.

Als externe Bibliothek zur Vereinfachung der Implementierung der Dateisystem-Komponente soll *FUSE* (vgl. FUSE) verwendet werden.

> With FUSE it is possible to implement a fully functional filesystem in a userspace program. (FUSE, Website)

Die Implementierung eines Anwendungsprogramms ist wesentlich weniger aufwändig als die Implementierung eines Moduls eines Betriebssystems, da das Betriebssystem als eine Art *virtuelle Maschine* die Basis für Anwendungsprogramme darstellt (vgl. Tanenbaum und Woodhull 1997, S. 1 ff.). Bei einem Anwendungsprogramm können Fehler in der Programmierung beispielsweise zum Absturz des Anwendungsprogrammes führen, in der Regel führt dies jedoch nicht zum Absturz des Systems. Fehler im Betriebssystem hingegen können zum Absturz des gesamten Systems führen, so dass das gesamte System neu gestartet werden muss. Es existieren zwar Betriebssysteme wie beispielsweise MINIX, bei denen Module als Anwendungsprogramm ausgeführt werden, diese stellen jedoch eher eine Ausnahme dar (vgl. Tanenbaum und Woodhull 1997, S. 14, S. 37 f.).

FUSE Architektur

In diesem Abschnitt soll die Architektur von *FUSE* betrachtet werden, um anschließend auf die Integration dieser Bibliothek in die C++ Implementierung einzugehen.

Bisher ist *FUSE* für verschiedene unix-artige Betriebsysteme, wie *Linux*, *Free-* und *NetBSD* sowie *Apple MacOS X* verfügbar. *FUSE* Dateisysteme werden, wie es unter unix-artigen Systemen üblich ist, in ein gewähltes Verzeichnis eingehängt, so dass alle Zugriffe auf dieses Verzeichnis vom Betriebssystemkern an das entsprechende Dateisystem delegiert werden.

Diese Plattformunabhängigkeit wird durch zwei Komponenten erreicht, die *FUSE* implementiert:

1. ein Betriebssytem-Modul

2. eine C Bibliothek

Wenn Anwendungen auf ein Verzeichnis zugreifen, in das ein *FUSE* Dateisystem eingehängt ist, werden die Zugriffe an das Betriebssystem-Modul weitergeleitet. Dieses wiederum leitet die Zugriffe mit Hilfe der C Bibliothek an die jeweiligen Funktionen der Implementierung weiter.

In Abbildung 6.1 (S. 82) wird der Aufruf `stat()`, der Informationen wie Dateigröße, Berechtigungen und andere Informationen ermittelt, resultierend aus einer Auflistung eines Verzeichnisses (`ls -l ~/docs`) beispielhaft dargestellt. Die Pfeile zeigen an, wie der Aufruf der Anwendung `ls` über die C Standardbibliothek an das Betriebsystem und anschließend an *FUSE* delegiert wird. *FUSE* reicht diesen Aufruf schlussendlich an das synchronisierende Dateisystem weiter, das diesen Aufruf implementiert und die benötigten Informationen ermittelt. Anschließend werden diese Informationen zurück an den ursprünglichen Aufrufer gesendet (gepunktete Pfeile). Der dunkel eingefärbte Knoten *synchronisierendes Dateisystem* stellt die zu implementierende Komponente dar, deren Komponenten bereits in Abbildung 5.8 (S. 58) gezeigt wurden. Die hell eingefärbten Knoten sind die Komponenten von *FUSE*.

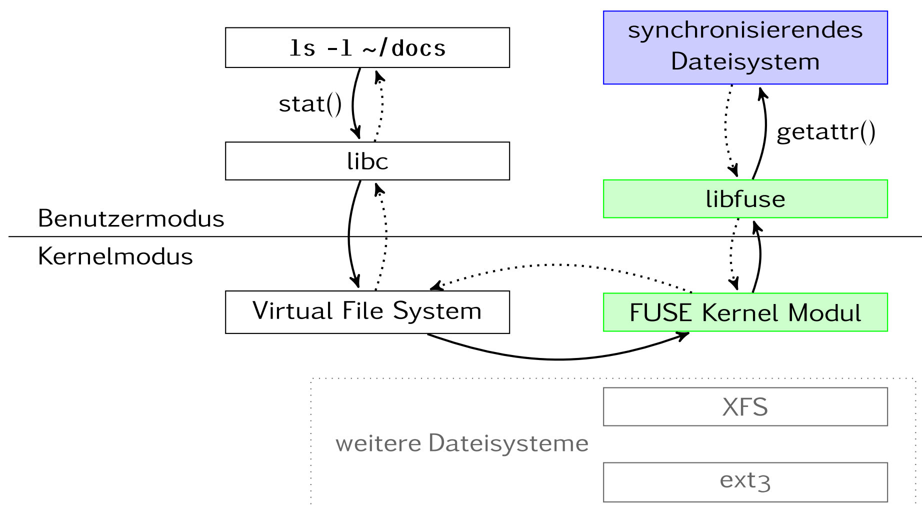

Abbildung 6.1.: Architektur von *FUSE* und Integration der Dateisystem-Schnittstelle

Die C Bibliothek von *FUSE* (libfuse) und das synchronisierende Dateisystem werden im Benutzermodus ausgeführt und können daher beliebige weitere Bibliotheken ver-

wenden, inklusive der C Standardbibliothek, deren Verwendung in der Abbildung 6.1 (S. 82) nicht berücksichtigt wurde.1.

Zunächst mögen diese vielen Weiterleitungen sehr aufwändig und zeitintensiv erscheinen, jedoch stellt dies in der Regel kein Problem dar, da die Kommunikation effizient erfolgt (vgl. FUSE). Erfolgt beispielsweise der Dateizugriff über ein Netzwerk, wie etwa bei *SSHFS* (vgl. SSH Filesystem), das den Zugriff auf ein entferntes Verzeichnis mit Hilfe von Open*SSH* (vgl. *Open*SSH) ermöglicht, dann ist in der Regel das Netzwerk der limitierende Faktor beim Dateizugriff und nicht die in Abbildung 6.1 (S. 82) dargestellte Kommunikation. Während der Entwicklung hat sich jedoch gezeigt, dass die transaktionale Protokollierung (siehe Kapitel 5.4.1 (S. 59)) wesentlich mehr Zeit benötigt als die Kommunikation zwischen dem *FUSE* Betriebssytem-Modul und der C Bibliothek.

Im Folgenden soll die Integration von *FUSE* und C++ erläutert werden. In diesem Zusammenhang wird auch die Verwendung von *FUSE* bei der Implementierung eines eigenen Dateisystems näher beleuchtet.

C++ Integration

Die Komponenten von *FUSE* sind in der Programmiersprache C implementiert, so dass eine Integration in C++ relativ einfach möglich sein sollte. Da die *FUSE* Bibliothek jedoch Zeiger auf Funktionen speichert (vgl. Kernighan und Ritchie 1990, S. 114 ff.), die die Funktionen der Dateisystem-Schnittstelle implementieren, ist eine Integration nicht ganz einfach. Insbesondere wenn ein Dateisystem als Klasse implementiert werden soll, ist die Integration etwas aufwändiger. Dennoch soll diese Möglichkeit in dieser Arbeit umgesetzt werden.

Die Implementierung eines eigenen Dateisystems mit Hilfe von *FUSE* ist relativ einfach. *FUSE* definiert eine Reihe von Funktionen, die eine Dateisystem-Schnittstelle nachbilden. Diese Funktionen werden von *FUSE* als Callback genutzt, die das *Command Pattern* (vgl. Gamma u. a. 1996, S. 273 ff.) implementieren.

> Sie können eine derartige Parametrisierung in einer prozeduralen Sprache mit einer **Callback-Funktion** erreichen. Dies ist eine Operation, die man irgendwo registriert und die zu einem späteren Zeitpunkt aufgerufen wird. (Gamma u. a. 1996, S. 276)

Jedes Dateisystem, das *FUSE* verwendet, muss einige dieser Funktionen implementieren und bei *FUSE* registrieren. Es müssen nicht alle Funktionen implementiert

werden, da nicht jede Funktion für ein bestimmtes Dateisystem sinnvoll sein muss. Erfolgen Zugriffe auf das Dateisystem, werden die registrierten Funktionen von *FUSE* aufgerufen.

Bei einem Callback wird also lediglich gespeichert, wie eine Funktion zu einem späteren Zeitpunkt aufgerufen werden kann. Die notwendigen Parameter werden in der Regel erst bei Aufruf des Callbacks übergeben.

Ein Callback in der Programmiersprache C ist lediglich ein Zeiger auf eine Funktion (vgl. Kernighan und Ritchie 1990, S. 114 ff.), der gespeichert und zu einem späteren Zeitpunkt aufgerufen wird. Funktionen eines Objekts können daher nicht als Zeiger auf eine Funktion gespeichert werden, da die Information über das zu verwendende Objekt nicht abgelegt werden kann. Außerdem kennt die Programmiersprache C die Objekte von C++ nicht und kann nicht mit diesen arbeiten. Trotzdem lassen sich diese Zeiger auf Funktionen mit C++ verwenden, indem Klassenfunktionen als Callback genutzt werden, da für diese kein konkretes Objekt existiert.

Mit Hilfe einer zusätzlichen Indirektion ist es dennoch möglich, Funktionen eines Objekts als Callback zu verwenden.

> Manchmal sagt man, dass es kein Problem in der Informatik gibt, das nicht durch eine weitere Indirektion gelöst werden kann. Obwohl das eine Übertreibung ist, enthält es ein Körnchen Wahrheit. (Tanenbaum 2003*b*, S. 935)

Die Klassenfunktion muss, wenn sie von *FUSE* aufgerufen wird, das konkrete Objekt ermitteln und für dieses die entsprechende Funktion aufrufen. Für diese Funktion bietet *FUSE* Unterstützung an, die verwendet werden soll.

FUSE ermöglicht es, einen Zeiger auf beliebige Daten – den sogenannte *user data pointer* – zu speichern und auf diesen innerhalb der Callback Implementierung zuzugreifen. Während der Initialisierung von *FUSE* kann dieser Zeiger genutzt werden, um auf das aktuelle Objekt zu verweisen. Da auf diesen Zeiger in jedem implementierten Callback zugegriffen werden kann, kann die Klassenfunktion, die den Callback darstellt, diesen Zeiger nutzen, um das aktuelle Objekt zu ermitteln. Anschließend kann so die Funktion des Objekts aufgerufen werden.

Wenn die Registrierung des Callbacks und der indirekte Aufruf in einer Basisklasse erfolgen, dann kann ein neues Dateisystem in C++ einfach implementiert werden, indem die neue Klasse von dieser Basisklasse erbt und die notwendigen Funktionen implementiert. Diese Klassenhierarchie ist am Beispiel des Aufrufs `getattr()` in Abbildung 6.2 (S. 85) dargestellt.

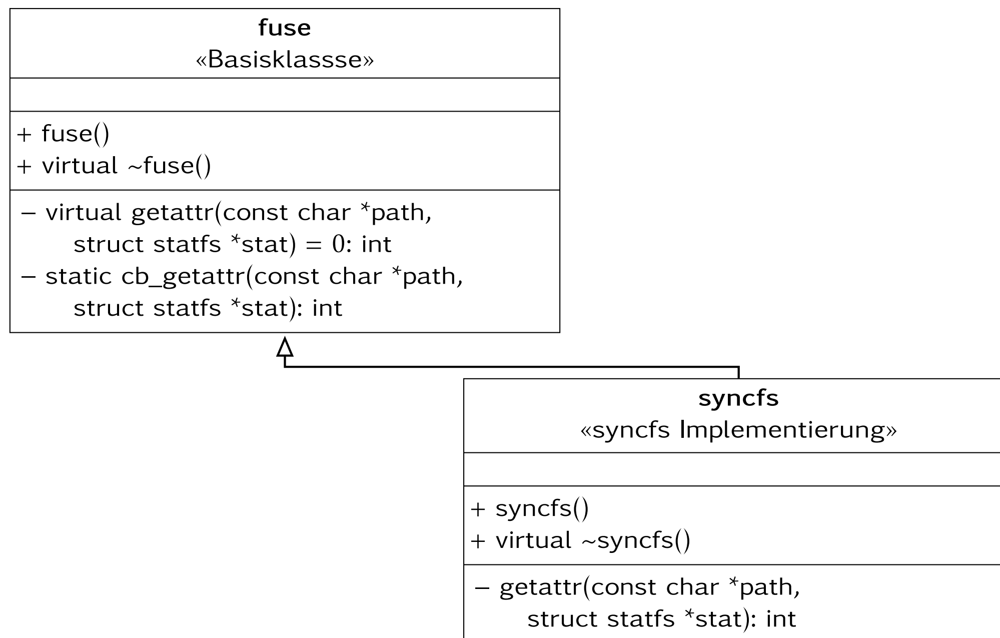

Abbildung 6.2.: Auszug *fuse* Basisklasse und konkrete, abgeleitete Implementierung *syncfs*

Zunächst soll die Basisklasse *fuse* betrachtet werden. Dort ist die private Klassenfunktion *cb_getattr()* definiert, die die Callback Funktion darstellt und die von *FUSE* aufgerufen wird. Diese stellt die zusätzliche Indirektion dar und greift auf den *user data pointer* zu, um das konkrete Objekt zu erhalten. Da dieser Zeiger vom Typ *void* ist, der einen unspezifischen Typ darstellt (vgl. Kernighan und Ritchie 1990, S. 91), muss eine Typumwandlung in den gewünschten Typ – die abgeleitete Klasse – erfolgen, um die gewünschte Funktion aufrufen zu können. Diese Typumwandlung erfolgt durch einen *C-style cast*, der in C++ nicht verwendet werden sollte, da keine Typprüfung möglich ist (vgl. Sutter und Alexandrescu 2005, S. 178, S. 180 f.). Dennoch stellt dies die einzige Möglichkeit dar, da keine Typinformationen vorhanden sind, um eine Typprüfung durchzuführen. Anschließend kann für dieses Objekt die Methode `getattr()` aufgerufen werden, die die konkrete Implementierung dieses Aufrufs der Dateisystem-Schnittstelle in der abgeleiteten Klasse enthält. Um die Schnittstelle, die die Nutzung von *FUSE* ermöglicht, und die konkrete Implementierung zu trennen, definiert die Klasse *fuse* private rein virtuelle Funktionen für alle auftretenden Callback Funktionen. Die Basisklasse kann so nicht mehr erzeugt werden und abgeleitete Klassen müssen die privaten rein virtuellen Funktionen implementieren.

Die Definition aller rein virtuellen Funktionen als private Funktionen nennt sich *Nonvirtual Interface (NVI)* Muster. Es trennt die öffentliche Schnittstelle und die spezifische Implementierung (vgl. Sutter und Alexandrescu 2005, S. 68 f.) und erfordert es, dass jede abgeleitete Klasse diese Funktionen implementiert. Die Funktionen müssen ebenfalls implementiert werden, wenn die Klasse nur indirekt von der Basisklasse abgeleitet ist.

Aus der Verwendung des *NVI* Musters ergibt sich ein großer Vorteil. Alle Aufrufe der implementierten Callbacks können nur duch *FUSE* oder die jeweilige Klasse erfolgen, niemals jedoch von einem Nutzer der Klasse. *FUSE* definiert verschiedene Invarianten und stellt einen Kontext zur Verfügung, wenn ein Callback aufgerufen wird. Eine Invariante ist beispielsweise, dass ein Pfad, auf den zugegriffen wird, immer ein absoluter Pfad ist. Als Kontext stellt *FUSE* beispielsweise die Kennung des aktuellen Anwenders und der verwendeten Anwendung zur Verfügung. Durch Verwendung des *NVI* Musters ist gewährleistet, dass diese Umgebung bei jedem Aufruf einer Callback Funktion existiert.

6.2.2. Protokollierung

In diesem Abschnitt wird die Komponente betrachtet, die die ausgeführten und modifizierenden Operationen protokolliert. Diese Daten werden von der Synchronisations-Komponente (Kapitel 6.2.4 (S. 94)) verwendet und an die Synchronisationspartner übermittelt.

Die Anforderungen an diese Komponente sind vielfältig (siehe Kapitel 5.4.1 (S. 59)). Diese Komponente soll transaktionsbasiert und gleichzeitig auch möglichst effizient arbeiten, da der Anwender die Auswirkungen dieser Komponente bei der Verwendung des synchronisierenden Dateisystems direkt erfährt. Arbeitet diese Komponente sehr langsam, dann muss der Anwender beispielsweise während des Kopierens oder Speicherns von Dateien entsprechend lange auf die Beendigung der Operation warten. Weiterhin ist es für die Synchronisation erforderlich, die Daten anhand verschiedener Kriterien selektieren zu können.

Insbesondere für die flexible Selektion und Aktualisierung bietet sich eine Datenbank zur Speicherung des Protokolls an. Für diese Aufgabe soll die *SQLite* Datenbank (vgl. SQLite Entwickler) verwendet werden. Diese besteht aus einer externen C Bibliothek und einem Kommandozeileninterpreter, so dass eine Integration sehr leicht möglich ist. Weiterhin bietet sie einen einfachen Zugriff auf die Daten und viele Funktionen, die für den Einsatz in dieser Implementierung sprechen.

> SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. (SQLite Entwickler, Website)

Die Operationen inklusive aller Parameter werden jedoch nicht in der Datenbank abgelegt, auch wenn nahezu alle Operationen aus sehr wenigen Daten bestehen. Die `write()` Operation jedoch kann auch aus sehr vielen Daten bestehen, je nachdem wieviele Bytes die Anwendung mit einem Aufruf der Operation schreiben möchte. Bei der Anwendung `cp` sind dies beispielsweise 4096B. Alle Operationen werden daher als Dateien abgelegt, so dass diese alle einheitlich verwendet werden können.

Für die Protokollierung werden fast alle in der Konzeption vorgestellen Informationen verwendet (siehe Kapitel 5.4.1 (S. 59)). Es wird jedoch nicht der Zeitpunkt, an dem die Operation ausgeführt wurde, gespeichert, sondern es wird ein Zähler verwendet, der mit jeder protokollierten Operation inkrementiert wird. Gerade wenn viele Operationen in sehr kurzer Zeit ausgeführt werden, kann dies zu *race conditions* (vgl. Tanenbaum 2003*b*, S. 118 f.) zwischen der Protokollierung und der Synchronisation führen. Diese Entscheidung wird stark von der Synchronisation (siehe Kapitel 6.2.4 (S. 94)) beeinflusst und daher in diesem Abschnitt erläutert.

Als zusätzliche Information wird die Kennung des Dateisystems protokolliert, so dass sich die Operationen jederzeit einem Dateisystem zuordnen lassen. Dies vereinfacht die Zusammenstellung der Operationen, die zu synchronisieren sind. Alle Daten werden in der Tabelle *vc_history* (siehe Tabelle 6.1 (S. 87)) abgelegt.

Information	Erklärung
`fs_uuid`	Dateisystem-Kennung
`path`	Pfad der protokollierten Operation
`tx_id`	Kennung der Operation
`version`	Version dieses Pfades, inkrementiert in Abhängigkeit des Pfades
`id`	globaler Zähler, inkrementiert mit jedem neuen Eintrag
`committed`	0 für vorbereitete und 1 für festgeschriebene Operationen

Tabelle 6.1.: Tabellendefinition der Tabelle *vc_history*

Operationen

Um die Operationen zu protokollieren, wurde eine Basisklasse implementiert, die alle notwendigen Informationen enthält und eine einfache Schnittstelle definiert. Dazu gehört das Setzen und Abfragen des Pfades der Operation und die Möglichkeit, ein

Objekt dieser Klasse als Bytestrom zu erhalten und umgekehrt, ein Objekt aus einem Bytestrom zu konfigurieren.

Jede Operation wird als eigene Klasse implementiert, die von dieser Basisklasse abgeleitet ist. Dies ermöglicht eine einfache Verwendung dieser Klassenhierarchie in einer Klon-Fabrik (vgl. Alexandrescu 2003, S. 283 ff.), die bei der Synchronisation benötigt wird. Eine Klon-Fabrik ist dem *Abstrakte Fabrik* Muster von Gamma u. a. (vgl. Gamma u. a. 1996, 10.7 ff.) sehr ähnlich. Um dynamisch Objekte der abgeleiteten Klassen – hier der Operationen – erzeugen zu können, werden Objekte als Prototyp in der Fabrik abgelegt. Später kann die Fabrik dieses Objekt anhand eines eindeutigen Schlüssels finden und benutzen, um ein neues Objekt des gewünschten Typs erzeugen zu lassen.

In Abbildung 6.3 (S. 89) ist die Klassenhierarchie exemplarisch dargestellt. Die Klasse `base_op` ist die Basisklasse und definiert die notwendigen Operationen und verwendet auch hier das *NVI* Muster. Dies hat den Vorteil, dass jede direkt oder indirekt von der Klasse `base_op` abgeleitete Klasse die entsprechenden privaten rein virtuellen Operationen implementieren muss.

Jede Operation wird durch einen Namen identifiziert und implementiert die Funktion `clone()`, die es ermöglicht, einen Klon des jeweiligen Objekts anzufertigen. Der Name wird für die Synchronisation (siehe Kapitel 6.2.4 (S. 94)) und die Klon-Fabrik benötigt, um die Zuordnung eines Bytestroms zu einem Objekt zu ermöglichen. Außerdem erhält jedes Objekt mit Hilfe einer *UUID* eine eindeutige Kennung, die ebenfalls für die Synchronisation benötigt wird. Zudem wird diese Kennung als Dateiname verwendet, wenn die Operation als Datei abgelegt wird. So lassen sich die Einträge in der Datenbank sehr einfach mit den zugehörigen Dateien verknüpfen, da die Kennung der Operation ebenfalls in der Datenbank abgelegt wird.

Um möglichst effizient und flexibel den Bytestrom nutzen zu können, wurde eine eigene Klasse implementiert, die außerdem alle Daten platformunabhängig im *Big Endian* Format speichert (vgl. Tanenbaum und van Steen 2008, S. 155 ff.). Das *Big Endian* Format wurde gewählt, weil dieses die sogenannte *Network Byte Order* ist (vgl. Rochkind 2004, S. 533), die jedes System in das sytem-eigene Format umwandeln kann, wenn es IP-basiert kommuniziert und die *Berkeley-Socket (BSD Socket)* Schnittstelle implementiert (vgl. Tanenbaum und van Steen 2008, S. 166 f.). So kann die Kommunikation auch zwischen Computern, die das *Little Endian* verwenden, und Computern, die das *Big Endian* Format verwenden, erfolgen.

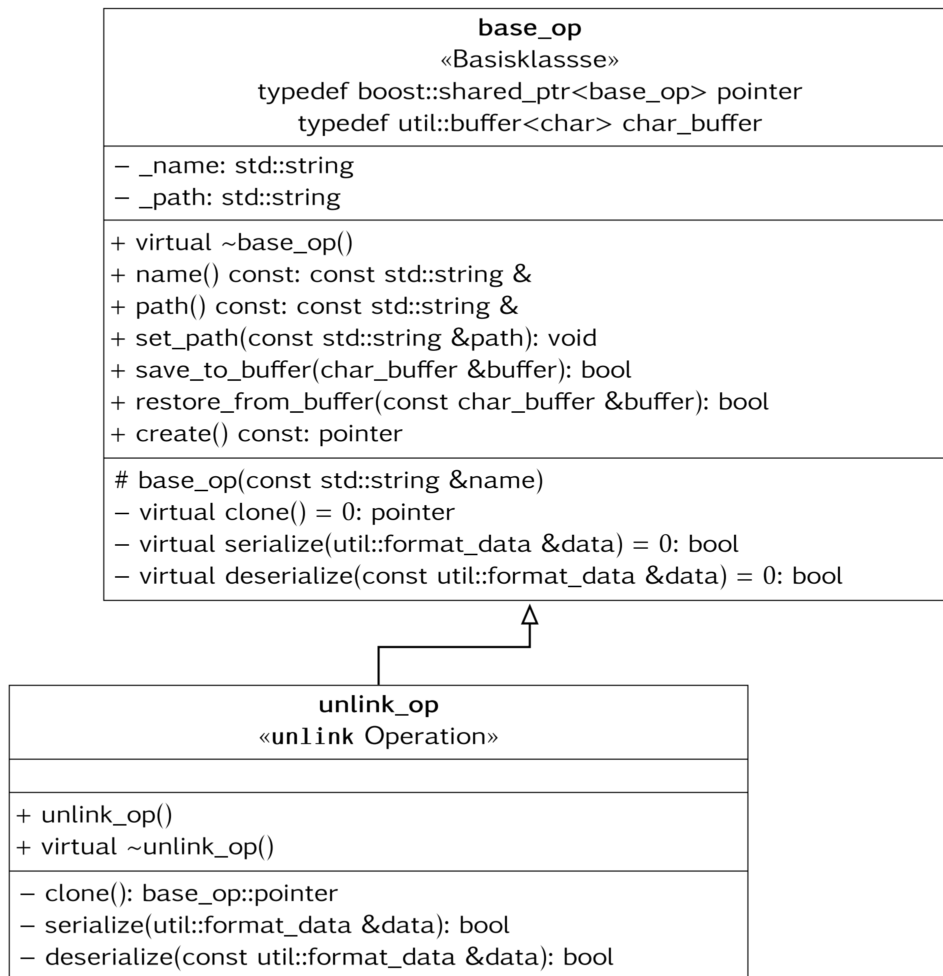

Abbildung 6.3.: Klassenhierarchie der Operationen

Bytestrom

Die Erzeugung des Bytestroms, implementiert in der Klasse *util::format_data*, der einzelnen Operationen ist relativ einfach möglich. Dazu könnten sogar Klassen der *C++ Standard Bibliothek (STL)* verwendet werden, da die Daten entweder ganze Zahlen oder Zeichenketten sind. Jedoch ist der resultierende Datenstrom nicht platformunabhängig. Außerdem werden keine Informationen über die Datentypen in den Bytestrom geschrieben. Zudem muss es möglich sein, jedes Byte – inklusive Steuerzeichen wie das `NUL`-Byte – zu lesen und zu schreiben. Diese Zeichen dürfen keine besondere Bedeutung besitzen, sie müssen wie jedes andere Zeichen behandelt werden. Daher wird eine Klasse implementiert, die den Zugriff auf die Daten

- versioniert,
- platformunabhängig und
- mit Datentypinformationen versehen

ermöglicht.

Insbesondere die Versionierung und Platformunabhängigkeit ist sehr wichtig. Die Versionierung ermöglicht es, das Format, in dem die Daten abgelegt werden, sehr flexibel anzupassen und zu erweitern. So ist es prinzipiell auch möglich, mehrere Formate aus Gründen der Kompatibilität zu unterstützen.

Durch die Speicherung der Typinformationen zu den einzelnen Daten wird zwar einerseits mehr Speicherplatz benötigt, andererseits besteht so aber die Möglichkeit, jeden Bytestrom zu analysieren, die Daten zu extrahieren und anzuzeigen. Außerdem kann eine Fehlerbehandlung erfolgen, falls versucht wird, einzelne Daten im falschen Format zu lesen.

Format

Das Format des Bytestroms ist sehr einfach gehalten, auch wenn dies an einigen Stellen mehr Speicherplatz benötigt, als unbedingt notwendig. Abbildung 6.4 (S. 91) veranschaulicht das Format.

Die Anzahl der Bytes für den Wert hängt vom jeweiligen Datentyp ab. Momentan werden folgende Datentypen unterstützt:

uint16_t: 2 Bytes, abgelegt als *Big Endian*

uint32_t: 4 Bytes, abgelegt als *Big Endian*

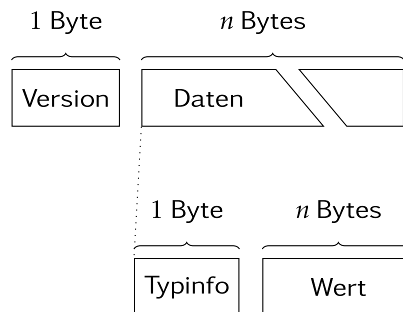

Abbildung 6.4.: Aufbau des Bytestroms

uint64_t: 8 Bytes, abgelegt als *Big Endian*

bool: 1 Byte, mit -1 für den Wert `true` und 0 für den Wert `false`

std::string: zusammengesetzter Typ

1. Länge der Zeichenkette, je nach Definition von `std::size_t` (auf 32-bit Systemen identisch mit `uint32_t`)
2. Daten der Zeichenkette bis zum ersten `NUL`-Byte

util::buffer<char>: zusammengesetzter Typ (Buffer Klasse)

1. Länge der Daten, je nach Definition von `std::size_t`
2. Daten

Das Lesen und Schreiben der einzelnen Werte wird über entsprechende Funktionen der Klasse ermöglicht. Um den Bytestrom in eine Datei zu schreiben oder über ein Netzwerk zu senden, kann eine Kopie des intern verwendeten Bytestroms erzeugt werden. Wird ein Bytestrom aus einer Datei ausgelesen oder über ein Netzwerk empfangen, kann umgekehrt dieser Bytestrom der Klasse übergeben werden, so dass über die Funktionen auf die enthaltenen Daten zugegriffen werden kann. Für die Verwaltung der rohen Daten, also des eigentlichen Bytestroms, wird eine spezielle Implementierung verwendet, die nachfolgend beschrieben wird.

Buffer Klasse

Um den Umgang mit Arrays effizient und flexibel zu gestalten, wurde eine spezielle Template-Klasse *util::buffer<>* entwickelt (vgl. Meyers 2006, S. 231 ff.). Diese erlaubt, einfache Datentypen wie beispielsweise `char`, `int` oder `double` als Array zu verwalten, und ermöglicht eine ähnliche Nutzung wie bei einem Array. Objekte können nicht

verwaltet werden, da es Ziel der Entwicklung ist, eine effiziente Möglichkeit zu schaffen, Arrays bestehend aus einfachen Datentypen zu verwalten. Der Vorteil, diese Klasse als Template-Klasse zu implementieren, liegt darin, dass der Quelltext nur einmal geschrieben werden muss, der Compiler jedoch typsicheren Code für jeden verwendeten Typ erzeugen kann und somit die Verwendung absichert. Dies wird in dem Listing 6.2 (S. 92) gezeigt. Bei Verwendung eines falschen Typs, erkennt der Compiler einen Fehler (Zeile 13, S. 92) und stoppt die Verarbeitung.

```
1  // Definition der beiden Typen
2  typedef util::buffer<double> double_buffer;
3  typedef util::buffer<char> char_buffer;
4
5  // Objekte erzeugen
6  double_buffer d;
7  char_buffer c;
8
9  double d_tmp[3] = { 42.0f, 43.0f, 44.0f };
10 char c_tmp[13] = "Hello␣World!";
11
12 d.assign(d_tmp, 3);  // erlaubter Aufruf
13 d.assign(c_tmp, 13); // Compiler erkennt die falsche Verwendung
```

Listing 6.2: Typsicherheit durch Verwendung von Templates

Die Implementierung eines eigenen dynamischen Arrays hat einige Vorteile, auch wenn Template-Klassen wie *boost::shared_array* die Verwendung von Arrays vereinfachen. *boost::shared_array* implementiert die *RAII (Resource Acquisition Is Initialization)* Strategie (vgl. Alexandrescu 2003, S. 24 f.) für Arrays, um einen Speicherbereich zu verwalten. Die Implementierung arbeitet analog zum *boost::shared_ptr*, verwaltet jedoch auch nur den Speicher, erlaubt aber nicht, die Größe effizient anzupassen.

Die Klasse *util::format_data*, die den Bytestrom verwaltet, könnte ein einfaches Array durchaus verwenden. Dennoch wäre die Verwaltung des Speichers sehr ineffizient, da das Array häufig minimal in der Größe angepasst werden muss, um neue Daten aufzunehmen. Außerdem ist der Einsatz einer optimierten Array Klasse auch an anderen Stellen notwendig und sinnvoll.

Die Allokierung von Speicher ist nicht ganz einfach und erfordert die Hilfe des Betriebssystems. Gerade wenn wiederholt die Speicherverwaltung genutzt wird, um Speicher zu allokieren und zu deallokieren, kostet die Ausführung unnötig Zeit (vgl. Alexandrescu 2003, S. 120 ff.). Um dieser Problematik entgegenzuwirken, kann der Speicher in größeren Blöcken allokiert werden, was jedoch einen erhöhten Speicher-

verbrauch zur Folge hat. Jedoch wird dadurch die Ausführungszeit merklich reduziert. Eine solche Strategie ist in der Template-Klasse *util::buffer*, die die Verwaltung des Arrays in der Klasse *util:format_data* übernimmt, implementiert.

Weiterhin nutzt die *util::buffer* Klasse die schon angesprochene *boost::shared_array* Klasse, um das Array zu verwalten. So kann eine *copy-on-write* Strategie (vgl. Tanenbaum 2003*b*, S. 261, S. 750) genutzt werden, so dass sich mehrere Objekte den gleichen Speicherbereich für das Array teilen. Dieser gemeinsame Zugriff erfolgt jedoch nur, solange lesend auf die Daten zugegriffen wird. Sobald ein Objekt die Daten des Array oder das Array selber verändern möchte, muss zunächst eine eigene Kopie des Arrays angefertig werden.

6.2.3. Dateizugriff

Der Dateizugriff erfolgt durch die existierenden Systemfunktionen. Da alle Dateien lokal abgelegt werden, muss lediglich das richtige Verzeichnis ausgewählt werden, in dem die Dateien und Verzeichnisse verwaltet werden. Anschließend können die entsprechenden Systemfunktionen aufgerufen werden.

Sowohl für Verzeichnisse als auch für Dateien wurde eine Adapter-Klasse implementiert, so dass die verwendeten Objekte leicht zwischengespeichert werden können. So kann beispielsweise auch sichergestellt werden, dass geöffnete Dateien bei Zerstörung des Objekts geschlossen werden.

Zudem existiert ein *Lock Manager*, der von dieser Komponente und der Komponente zur Synchronisation genutzt wird, da nicht beide Komponenten zeitgleich die gleiche Datei oder das gleiche Verzeichnis modifizieren dürfen. Dieser *Lock Manager* ist lediglich eine Liste, die jeden Eintrag maximal einmal enthält, und die von den beiden genannten Komponenten gemeinsam genutzt wird. Bevor ein Zugriff auf eine Datei oder ein Verzeichnis erfolgt, wird versucht das entsprechende Objekt, das den Zugriff auf die Datei oder das Verzeichnis kapselt, einzutragen. Gelingt dies, kann die Datei oder das Verzeichnis modifiziert werden. Der *Lock Manager* blockiert dabei den Aufrufer nicht. Für die Komponente zur Synchronisation wäre diese Blockade möglicherweise noch akzeptabel, für den Dateizugriff jedoch nicht. Der Zugriff auf das Dateisystem darf nicht blockieren, da dies auch direkt Auswirkungen auf die Anwendung hat, die gerade den Zugriff ausführt. Zudem wäre dieses Verhalten vom Anwender nicht erwartet und nicht unbedingt nachvollziehbar.

Weiterhin wird der Zugriff auf das Verzeichnis *.syncfs.d* unterbunden, da dieses die protokollierten Operationen und weitere Daten enthält (siehe Kapitel 6.3 (S. 99)).

6.2.4. Synchronisation

Die Synchronisation erfolgt immer im Hintergrund, parallel zur Nutzung des synchronisierenden Dateisystems, das heißt, das synchronisierende Dateisystem muss eingebunden werden, damit eine Synchronisation erfolgen kann.

In Kapitel 5.4.1 (S. 59) wurde als zusätzliche Information zu der Version auch immer der Zeitpunkt, an dem die Operation ausgeführt wurde, protokolliert und später zur Synchronisation herangezogen. Wenn der verwendete Zeitstempel eine maximale Auflösung von einer Sekunde hat, dann besteht die Möglichkeit, dass *race conditions* (vgl. Tanenbaum 2003*b*, S. 118 f.) auftreten. Wenn eine Synchronisation startet und alle seit dem Zeitpunkt der letzten Synchronisation hinzugekommenen Operationen ausgewählt werden, gehören auch Operationen dazu, die kurz vor dem Start ausgeführt wurden. Wird beispielsweise gerade das Dateisystem aktiv genutzt und viele Operationen in kurzer Zeit ausgeführt, dann könnte der Fall eintreten, dass viele Operationen den gleichen Zeitstempel tragen. Eine parallel laufende Synchronisation könnte nun bei der Auswahl der Operationen nur einige dieser Operationen auswählen, aber nicht alle. Außerdem lässt sich durch Operationen, die den gleichen Zeitstempel besitzen, nicht mehr die ursprüngliche Reihenfolge rekonstruieren.

Daher soll in der Implementierung – wie schon bei der Erläuterung zur Protokollierung (Kapitel 6.2.2 (S. 86)) angesprochen – anstelle des Zeitstempels ein Zähler verwendet werden, der mit jeder Operation unabhängig vom Pfad inkrementiert wird. Dies verhindert die zuvor beschriebene *race condition* und sorgt ebenfalls dafür, dass die Operationen in der Reihenfolge ausgeführt werden können, die ursprünglich verwendet wurde. Die erste protokollierte Operation ist 1, 0 wird als Wert für jeden Synchronisationpartner verwendet, mit dem noch keine Synchronisation erfolgt ist.

Dieser Zähler ist *nicht* auf allen Dateisystemen identisch, sondern definiert nur die Reihenfolge, in der die Operationen auf dem aktuellen Dateisystem ausgeführt worden sind. Wenn nun also während einer Synchronisation Operationen von anderen synchronisierenden Dateisystemen empfangen werden, werden diese nach erfolgreicher Ausführung ebenfalls im Protokoll festgehalten. Dies ist notwendig, damit diese Operationen ebenfalls synchronisiert werden. Ansonsten wäre eine Verkettung mehrer Synchronisationspartner, wie sie in Beispiel 8 (S. 51) (Kapitel 5.2 (S. 50)) beschrieben wurde, nicht möglich. Zu jeder Operation wird zudem die Dateisystem-Kennung festgehalten, so dass die Operation dem Dateisystem zugeordnet werden kann, auf dem sie ursprünglich ausgeführt wurde. So können die zu übertragenden Operationen in Abhängigkeit des Empfängers stärker eingeschränkt werden.

Weiterhin ist es notwendig, dass der Zugriff auf Dateien und Verzeichnisse exklusiv erfolgt, wenn die Operationen ausgeführt werden. Es darf auf die jeweilige Datei oder das jeweilige Verzeichnis nicht gleichzeitig durch diese Komponente und den Anwender über die Dateizugriffs-Komponente zugegriffen werden, da dies zu unvorhersehbaren Ergebnissen führen und zu defekten Dateien führen kann. Wie dieser exklusive Zugriff realisiert wird, ist in Kapitel 6.2.3 (S. 93) erläutert.

Die von anderen Dateisystemen empfangen Operationen werden in einem separaten Bereich abgelegt. Die Operationen werden analog zur Protokollierung (siehe Kapitel 6.2.2 (S. 86)) in einem Verzeichnis abgelegt. Zusätzlich werden die Version, der Wert des globalen Zählers, die Kennung der Operation und des Dateisystems, von dem diese Operation empfangen wurde, in einer Datenbank in der Tabelle *vc_sync* (siehe Tabelle 6.2 (S. 95)) abgelegt, so dass eine Zuordnung zwischen der Operation und dem Datenbankeintrag möglich ist. Diese Komponente kann nun alle empfangenen Operationen in der Reihenfolge des anderen Dateisystems auf dem lokalen Dateisystem erneut ausführen. Dazu wird zunächst der globale Zähler des entfernten Dateisystems ausgelesen. Anschließend können alle Operationen zusammengestellt werden, um sie auszuführen. Wenn alle Operationen ausgeführt wurden, kann der globale Zähler der letzten Operation in der Konfiguration für das entsprechende Dateisystem hinterlegt werden.

Sollten Operationen während der Ausführung fehlen, also aufgrund des globalen Zählers eine Lücke entdeckt werden, so muss die Ausführung abgebrochen werden. Dies wird jedoch schon vorher überprüft, um Probleme zu vermeiden. Eine fehlende Operation könnte beispielsweise eine `write()` Operation von vielen sein. Würden nun die die Operationen bis zur fehlenden Operation ausgeführt werden, wäre die Datei in einem inkonsisten Zustand.

Operationen können fehlen, weil der Sender der Operationen diese nicht in der Reihenfolge, die sich aus dem globalen Zähler ergibt, senden muss.

`Information`	**Erklärung**
`fs_uuid`	Dateisystem-Kennung des Senders
`path`	Pfad der protokollierten Operation
`tx_id`	Kennung der Operation
`version`	Version dieses Pfades, inkrementiert in Abhängigkeit des Pfades
`id`	globaler Zähler, des entfernten Dateisystems

Tabelle 6.2.: Definition der Tabelle *vc_sync*

Kommunikation

Die Kommunikation erfolgt über eine TCP/IP-Verbindung, so dass sichergestellt ist, dass die Daten vollständig und in der richtigen Reihenfolge empfangen werden können (vgl. Tanenbaum 2003*a*, S. 580 ff.). Um die Implementierung zu vereinfachen, wird *boost::asio* verwendet. Diese Bibliothek ermöglicht sehr einfach, objektorientiert und flexibel die Implementierung von Netzwerkservern und Clients.

boost::asio implementiert das *Proactor* Muster (vgl. Schmidt u. a. 2004, S. 215 ff.) und ermöglicht so eine einfache Implementierung, um Daten asynchron zu senden oder zu empfangen. Mit Hilfe von *boost::bind* ist es möglich, jede beliebige Funktion einer Klasse an *boost::asio* zu übergeben, so dass diese das Ergebnis des asynchronen Aufrufs erhält. Dies Verhalten wird in dem Listing 6.3 (S. 96) beispielhaft gezeigt.

```
void connection::start() {
  namespace as = boost::asio;

  /** Daten lesen oder auf neue Daten warten... */
  as::async_read(m_socket,
                 as::buffer(m_buffer.get(), m_buf_size),
                 as::transfer_at_least(128),
                 boost::bind(&connection::handle_read,
                             this,
                             as::placeholders::error,
                             as::placeholders::bytes_transferred));
}

void connection::handle_read(const boost::system::error_code &e,
                             std::size_t bytes_transferred) {
  /** Verarbeitung der Daten... */
  bool result = parse_data(m_buffer, bytes_transferred);

  /** Weitere Daten lesen oder auf neue Daten warten... */
  as::async_read(m_socket,
                 as::buffer(m_buffer.get(), m_buf_size),
                 as::transfer_at_least(128),
                 boost::bind(&connection::handle_read,
                             this,
                             as::placeholders::error,
                             as::placeholders::bytes_transferred));
}
```

Listing 6.3: Beispiel für das asynchrone Empfangen von Daten über eine Netzwerkschnittstelle mit *boost::asio*

boost::asio verwendet einen I/O-Service, der die Ereignisbehandlung kapselt. Alle Funktionen, die an asynchrone Funktionsaufrufe gebunden werden, wie in Beispiel 6.3 (S. 96) (Zeilen 8 und 23), werden im gleichen Thread aufgerufen, in dem der I/O-Service ausgeführt wird. Dies ermöglicht die Verwendung der *boost::asio* Bibliothek inklusive asynchroner Funktionalität, so als würde die Anwendung synchron laufen. Es müssen keinerlei Vorkehrungen für eine multi-threaded Ausführung getroffen werden, wie beispielsweise die Verwendung von Sperren zum gegenseitigen Ausschluss beim Zugriff auf Objektvariablen (vgl. Tanenbaum und Woodhull 1997, S. 58 ff.).

Die ausgetauschten Nachrichten verwenden das schon in Kapitel 6.2.2 (S. 90) vorgestellte Format. Da Daten bei einer TCP/IP-Verbindung als Strom empfangen werden, wird die Verarbeitung der Daten vereinfacht, wenn die Nachricht aus einem definierten Kopf- und einem variablen Datenteil besteht. Der Kopfteil besteht bei jeder Nachricht aus den gleichen Informationen und hat eine feste Länge. Innerhalb des Kopfteils kann dann die Länge des variablen Datenteils hinterlegt werden, so dass nach dem Empfang des Kopfteils feststeht, wieviel Bytes noch zu empfangen sind, um die Nachricht zu dekodieren.

Der Kopfteil besteht aus einer Identifizierung, der Version des Formats, dem Nachrichtentyp und der Länge des Datenteils. Wenn die Nachricht erzeugt wird, wird an den Kopfteil der Datenteil angehängt. Anschließend wird über diesen Teil ein kryptografischer Hash-Wert berechnet und an die Nachricht angefügt. Zur Berechnung des kryptografischen Hash-Wertes wird die Bibliothek *libgcrypt* des *GnuPG* Projekts (vgl. GnuPG Kryptografie Bibliothek) verwendet. Dort sind verschiedene Algorithmen implementiert, von denen momentan *SHA256* zur Berechnung verwendet wird, da dieser Algorithmus bisher kollisionsresistent ist (vgl. Schmeh 2007, S. 214 ff.).

Der Datenteil der Nachricht ist abhängig vom Typ der Nachricht. Die folgenden Nachrichtentypen werden für eine Synchronisation verwendet:

announce: Dieser Nachricht übermittelt Informationen des Dateisystems. Dazu zählen eine eindeutige Kennung (*UUID*) und Informationen, wie eine Kommunikation erfolgen kann. Diese Nachricht muss als erste Nachricht bei jeder neuen Kommunikation (IP-Adresse und Port) erfolgen, um die Operationen zuordnen zu können.

Ist das Dateisystem noch nicht bekannt, so wird es einer Liste hinzugefügt, damit es zukünftig als bekannt angesehen wird und eine Synchronisation erfolgen kann.

synchronise: Diese Nachricht übermittelt die ausgeführten Operationen, die fast alle notwendigen Daten schon beinhalten. Zusätzlich muss noch die Version und der globale Zähler der Nachricht hinzugefügt werden, um die Reihenfolge der Operationen ermitteln zu können. Der globale Zähler ist nur notwendig, um die ursprüngliche Reihenfolge der Operationen zu erhalten.

acknowledge: Eine *acknowledge* Nachricht wird als erfolgreiche Bestätigung gesendet, wenn eine Operation empfangen und temporär abgelegt wurde.

negative acknowledge: Für den Fall, dass eine empfangene Operation nicht abgelegt werden konnte, wird diese Nachricht geschickt und der Empfänger kann die Nachricht erneut erneut senden, eventuell auch zu einem späteren Zeitpunkt.

Als erstes muss eine *announce* Nachricht empfangen werden, damit die folgenden Nachrichten dem entsprechenden Dateisystem zugeordnet werden können. Dies ist für die spätere Implementierung der Konflikterkennung und -lösung notwendig, da Konflikte immer nur zwischen zwei Synchronisationspartner bestehen. Wenn nun ein Dateisystem mit mehreren anderen Dateisystemen synchronisiert wird, kann gegenüber einem Dateisystem ein Konflikt bestehen, gegenüber den anderen kann jedoch alles konfliktfrei synchronisiert werden. Mit Hilfe der Dateisystem-Kennung lässt sich so entscheiden, welche Operationen ausgeführt werden dürfen und welche nicht.

Nach Empfang einer *synchronise* Nachricht wird diese an eine Klasse übergeben, die die Operation entpackt und in einem temporären Bereich ablegt. War dies erfolgreich, wird eine *acknowledge* Nachricht an den Aufrufer zurückgegeben, der diese an den Sender – das andere Dateisystem – sendet. War die Ablage der empfangenen Nachricht nicht erfolgreich, wird eine *negative acknowledge* Nachricht zurückgegeben und gesendet. Die Operation kann dann sofort oder zu einem späteren Zeitpunkt erneut gesendet werden.

Wenn eine Synchronisation erfolgen soll, wird versucht eine schon bestehende Verbindung zu dem gewünschten Dateisystem zu nutzen, so dass für eine bidirektionnale Synchronisation immer nur eine Verbindung existiert. Entsprechend könnte nach Empfang einer *announce* Nachricht die Synchronisations-Komponente auch benachrichtigt werden, dass gerade *synchronise* Nachrichten Empfangen werden, um so ebenfalls die lokal erzeugten Operationen zu synchronisieren. Dies ist aktuell jedoch nicht implementiert.

6.3. Aktueller Stand der Entwicklung

In diesem Kapitel soll der aktuelle Stand der prototypischen Implementierung beschrieben werden. Die Implementierung erlaubt das Synchronisieren von Dateien und Verzeichnissen, ohne jedoch Konflikte erkennen oder lösen zu können.

Um das synchronisierende Dateisystem zu nutzen, muss es in ein Verzeichnis eingehängt werden. Auf die in diesem Verzeichnis enthaltenen Objekte kann dann nicht zugegriffen werden. Daher muss ein weiteres Verzeichnis existieren, in dem die Objekte und die Verwaltungsinformationen abgelegt werden. In Listing 6.4 (S. 99) wird das synchronisierende Dateisystem in das Verzeichnis *sync* eingehängt. Die Objekte und die Verwaltungsinformationen werden in dem Verzeichnis *storage* abgelegt. Der Zugriff erfolgt dann über das Verzeichnis *sync*. Ein großer Nachteil ist, dass weiterhin über das Verzeichnis *storage* auf die Objekte zugegriffen werden kann. Diese Zugriffe werden auch durch das synchronisierende Dateisystem nicht wahrgenommen. Eine Möglichkeit das zu verhindern, existiert nicht.

```
1 [user@notebook]:~/> bin/syncfs storage sync
```

Listing 6.4: Einhängen eines synchronisierenden Dateisystems

Das synchronisierende Dateisystem selbst enthält alle Informationen, die für den Betrieb erforderlich sind. Keine Verwaltungsdaten werden außerhalb des Dateisystems abgelegt. In dem Verzeichnis *.syncfs.d* werden neben der Konfiguration auch die protokollierten und von anderen synchronisierenden Dateisystemen empfangen Operationen abgelegt.

Zur Speicherung der Konfiguration wird, wie auch zur Verwaltung des Protokolls, eine *SQLite* Datenbank verwendet, da so die Konfiguration komfortabel geschrieben, gelesen und aktualisiert werden kann. Bei der Nutzung eines synchronisierenden Dateisystems wird überprüft, ob eine Konfiguration vorliegt. Falls keine Konfiguration vorliegt, wird eine erzeugt und die Parameter werden mit Standardwerten belegt. Die Konfiguration wird in einer Tabelle abgelegt, die lediglich aus einem Schlüssel – dem Parameternamen – und einem Wert besteht. Momentan sind folgende Parameter möglich, deren Name allerdings zum besseren Verständnis angepasst wurde:

Dateisystem-Kennung: Die Kennung des Dateisystems. Hierfür wird eine *UUID* verwendet, um möglichst eindeutige Werte zu erhalten.

Der Standardwert bei der Initialisierung wird generiert, so dass jedes Dateisystem eindeutig gekennzeichnet ist.

Transkationsverzeichnis: In diesem Verzeichnis, das sich unterhalb des Verzeichnisses *.syncfs.d* befindet, werden alle protokollierten Operationen abgelegt.

Als Standardwert wird der Verzeichnisname *tx* verwendet.

temporäres Transaktionsverzeichnis: In diesem Verzeichnis, das ebenfalls unterhalb des Verzeichnisses *.syncfs.d* liegt, werden alle empfangen, aber noch nicht ausgeführten Operationen abgelegt.

Als Standardwert wird das Verzeichnis *sync_tx* verwendet.

Host Adresse: Der Name oder die IP-Adresse, auf der auf eingehende Verbindungen anderen synchronisierender Dateisysteme gewartet wird.

Als Standardwert wird hier *localhost* verwendet, so dass keine Verbindungen anderer Computer möglich sind.

Port: Für eingehende TCP/IP-Verbindungen wird zusätzlich ein Port benötigt. Dieser wird in diesem Parameter gespeichert.

Als Standardwert wird 4242 verwendet, so dass jeder Anwender ohne spezielle Berechtigungen diesen Port verwenden kann. Alle Ports unterhalb 1024 erfordern spezielle Berechtigungen.

Versionsdatenbank: Der Name der Datenbank, die die Tabellen *vc_history* und *vc_sync* enthält.

Um Erweiterungen leichter zu integrieren, sollte die Konfigurationstabelle für die anderen verwendeten Tabellen eine Versionsnummer speichern. Neuere Versionen können dann die Daten der älteren Version lesen und die Struktur gegebenenfalls anpassen.

Es existiert zusätzlich eine Tabelle (*peers*), die alle notwendigen Informationen der Synchronisationspartner enthält. Diese Tabelle enthält zunächst keine Einträge. Soll eine Synchronisation mit einem Synchronisationspartner erfolgen, muss der Anwender die entsprechenden Informationen (Dateisystem-Kennung, Host Adresse und Port) eintragen. Anschließend werden die Operationen mit diesem synchronisierenden Dateisystem ausgetauscht. Momentan muss dazu das synchronisierende Dateisystem neu eingebunden werden, da die Tabelle nur einmal während des Startens gelesen wird.

Wird nun eine Verbindung von einem synchronisierenden Dateisystem aufgebaut und eine *announce* Nachricht gesendet, werden die in dieser Nachricht enthaltenen Daten (Dateisystem-Kennung, Host Adresse und Port) automatisch in die Tabelle eingetragen, falls noch kein Eintrag existiert. Somit ist es ausreichend, die Verbindung auf einem synchronisierenden Dateisystem einzurichten. Die nachfolgende Aufzählung stellt zusammenfassend alle in dieser Tabelle enthaltenen Parameter dar:

Host Adresse: Der Name oder die IP-Adresse des Synchronisationspartners.

Port: Der TCP/IP-Port, auf dem der Synchronisationspartner auf eingehende Verbindungen wartet.

Zähler: Der globale Zähler, der der letzten synchronisierten Operation entspricht. Erfolgte noch keine Synchronisation mit dem Synchronisationspartner, wird als Standardwert eine 0 eingetragen, so dass alle Operationen übertragen werden.

Dateisystem-Kennung: Die Dateisystem-Kennung des Synchronisationspartners.

Zugriffe auf das Verzeichnis *.syncfs.d* über das synchronisierende Dateisystem sind nicht erlaubt und werden unterbunden. Es wird ein entsprechender Systemfehler zurückgeliefert, dass der Zugriff aufgrund fehlender Berechtigungen nicht gestattet ist. Alle Dateiinformationen sind relativ zum Wurzelverzeichnis des synchronisierenden Dateisystems gespeichert, so dass der Anwender das syqnchronisierende Dateisystem beliebig einbinden kann.

Wurde ein synchronisierendes Dateisystem eingebunden, so versucht dieses nach kurzer Wartezeit, die protokollierten Operationen an die eingetragenen Synchronisationspartner zu übermitteln. Momentan passt sich die Synchronisation noch nicht an die Umgebung an. Wenn also festgestellt wird, dass ein Synchronisationspartner nicht erreichbar ist, wird trotzdem weiterhin im üblichen Rhythmus versucht, diesen Synchronisationspartner zu konktaktieren. Sinnvoller wäre es, für nicht erreichbare Synchronisationspartner die Wartezeit zu erhöhen. Dies könnte auch in Abhängigkeit der fehlgeschlagenen Verbindungsversuche geschehen.

Zusammenfassend soll der Status der Implementierung der einzelnen Komponenten dargestellt werden.

Obwohl die Synchronisation und Protokollierung implementiert ist, besteht viel Potential zur Verbesserung und Optimierung der Implementierung.

Komponente	vollständig implementiert?
Protokollierung	✓
Synchronisation	✓
Konflikterkennung	✗
Konfliktlösung	✗
Anwenderschnittstelle	✗

Tabelle 6.3.: Übersicht der implementierten Komponenten des synchronisierenden Peer-to-Peer Dateisystems

6.3.1. Optimierungsmöglichkeiten

Es sollen in diesem Abschnitt noch einige Probleme und mögliche Optimierungen genannt werden, die während der Implementierung und manueller Tests deutlich geworden sind.

Die Protokollierung wurde, wie in Kapitel 5.4.1 (S. 59) vorgestellt, implementiert. Während der Implementierung und manueller Tests hat sich gezeigt, dass diese direkte Umsetzung bei vielen Änderungen innerhalb eines kurzen Zeitraums die Dateizugriffe stark verzögert. Das Entpacken eines Archivs etwa dauert sehr viel länger, wenn die Dateien auf das synchronisierende Dateisystem geschrieben werden. Als Optimierung wäre eine eine asynchrone Protokollierung im Hintergrund möglich, durch die sich einerseits die Gefahr erhöhen würde, dass Protokoll und Dateien beziehungsweise Verzeichnisse nicht konsistent sind, andererseits würden sich aber auch die Dateizugriffe beschleunigen.

Da die Operationen zustandslos protokolliert werden (siehe Kapitel 5.4.1 (S. 59)), werden sehr viele Operationen erzeugt. Für jede Operation werden einge Informationen in der verwendeten Datenbank abgelegt. Zusätzlich werden alle für die Operation notwendigen Daten, die den ursprünglichen Parametern des Funktionsaufrufs entsprechen, in einer Datei abgelegt. Das folgende Beispiel verdeutlicht wie schnell die Anzahl der in Dateien abglegten Operationen steigen kann.

Beispiel 11 *Wird mit der Anwendung* **cp** *eine* 3MiB[6] *große Datei auf das synchronisierende Dateisystem kopiert, so ergeben sich 768 Operationen. Es werden also auch 768 Dateien erzeugt. Diese Anzahl hängt in hauptsächlich damit zusammen, dass* **cp** *Dateien in* 4096B *großen Blöcken kopiert.*

[6] Als Einheiten werden hier Binärpräfixe anstelle der SI-Einheiten verwendet (siehe `http://de.wikipedia.org/wiki/Binärpräfix`). Die Binärpräfixe verwenden Zweierpotenzen anstelle von Zehnerpotenzen. 1 MiB (2^{20}B) entspricht 1024 KiB und wiederum 1 KiB (2^{10}B) entspricht 1024 B.

Zunächst wird die Dateigröße in Byte umgerechnet:

$$\begin{aligned} 3\text{MiB} &= 3 \cdot 1024\text{KiB} \\ &= 3 \cdot 1024 \cdot 1024\text{B} \\ &= 3145728\text{B} \end{aligned}$$

Anschließend kann die Anzahl der Blöcke ermittelt werden:

$$\frac{3145728\text{B}}{4096\text{B}} = 768 \textit{ Blöcke}$$

*Bei der Digitalisierung von 10 Audio-CDs ergeben sich beispielsweise 150 Dateien von durchschnittlich 3*MiB*. Als Folge werden somit 115200 Dateien beziehungsweise Operationen für diese 150 Dateien erzeugt.*

Als sinnvolle Optimierung sollten `write()` Operationen in so einem Fall nicht zustandslos abgelegt werden. Dadurch könnte die Anzahl der Operationen sehr stark reduziert werden. Dies könnte auch die Protokollierung beschleunigen, da nur noch eine einzige Operation anzulegen ist.

Als problematisch hat sich dieses Verhalten ebenfalls gezeigt. Die prototypische Implementierung sendet alle neuen Operationen direkt an die Synchronisationspartner. Da die Protokollierung aus den genannten Gründen den Dateizugriff bremst, kann es beim Kopieren größerer Dateien, die mehrere hundert Operationen erzeugen, dazu führen, dass einzelne Operationen bereits synchronisiert und erneut ausgeführt worden sind, obwohl die gesamte Datei noch nicht kopiert wurde. An dieser Stelle sind noch einige Verbesserung notwendig, die auch die Protokollierung beschleunigen. Durch das bereits genannte Berücksichtigen der Zustände beim Kopieren von Dateien, würde das Festschreiben der Operation solange zurückgehalten werden, bis die Datei vollständig geschrieben würde. So können auch keine partiell vorhandenen Dateien synchronisiert werden.

6.4. Testen der Implementierung

In diesem Abschnitt wird beschrieben, wie ein synchronisierendes Dateisystem getestet werden kann. Nach Möglichkeit sollten alle Tests automatisch erfolgen, damit sichergestellt werden kann, dass die Tests immer gleich ablaufen und die Ergebnisse reproduzierbar sind (vgl. Hunt und Thomas 2003, S. 218 ff.).

Die Implementierung erfolgte in Anlehnung an die Vorgehensweise des *Extreme Programming* (vgl. Beck 2003). Es wurden jedoch nicht so konsquent Tests implementiert, wie dies empfohlen wird. Für viele Klassen, die Basisfunktionalität implementieren, existieren Testfälle. Die Integration verschiedener Klassen ist jedoch nur marginal durch Tests abgedeckt.

Es gibt verschiedene Ebenen, auf denen das synchronisierende Dateisystem getestet werden kann. Die folgende Liste stellt Tests der verschiedenen Ebenen dar, angefangen mit der untersten Ebene:

1. Unit-Tests
2. Integrationstests
3. Systemtest

Die Unit-Tests werden verwendet, um die korrekte Funktionalität der implementierten Klassen zu überprüfen. Teilweise verwenden einzelne Test, wie der Test der Klon-Fabrik, andere Klassen, so dass nicht immer die Funktionalität nur einer Klasse getestet wird. Viele Tests verwenden zudem auch eine *Log*-Komponente, um Ausgaben im Fehlerfall auszugeben.

Es sind nur sehr wenige Integrationstests, die das Zusammenwirken mehrerer Komponenten testen, vorhanden. Der Test der Klon-Fabrik könnte dieser Kategorie zugeordnet werden, ebenso ein einfacher Test, um eine Synchronisation mit einem synchronisierenden Dateisystem zu initiieren. Diese Tests sind jedoch nicht sehr ausgereift.

Ein Systemtest, der das gesamte System in Betrieb als *Black-Box* – also ohne Kenntnis von Implementierungsdetails – testet, existiert ebenfalls nur ansatzweise. Dieser Test deckt im Wesentlichen die vorhandenen Möglichkeiten der einfachen Synchronisation ab. Bisher wurde dieser Test nur manuell ausgeführt, wodurch die Ergebnisse nur bedingt reproduzierbar sind.

Für den Systemtest werden zunächst zwei synchronisierende Dateisysteme eingebunden. Beide Dateisysteme dürfen nicht vorkonfiguriert sein, sondern müssen ihre Standardkonfiguration erzeugen. Anschließend muss ein Dateisystem wieder beendet werden, um die Konfiguration anpassen zu können. Da beide Dateisysteme den gleichen IP-Port für die Kommunikation verwenden, muss dieser angepasst werden. Zudem muss die Synchronisation mit dem anderen synchronisierenden Dateisystem konfiguriert werden. Nachdem anschließend das Dateisystem erneut eingebunden wird, können Dateien in das Dateisystem kopiert werden. Die aktuelle Implementierung

synchronisiert diese Operationen sobald diese festgeschrieben worden sind. Abschließend können für die kopierten und die synchronisierten Dateien kryptografische Hash-Werte, beispielsweise unter Verwendung des SHA256 Algorithmus', berechnet werden. Wenn diese für die entsprechenden Dateien identisch sind, funktioniert die Synchronisiation. Dieser Test stellt zwar einen sehr einfachen Test dar, testet aber die synchronisierenden Dateisystem ausschließlich aus Sicht des Anwenders. Wissen über Implementierungsdetails existieren nicht.

Im Zuge einer Weiterentwicklung ist es zwingend erforderlich, die Anzahl der Tests auf allen genannten Ebenen zu erhöhen, so dass auch Integrations- und Systemtests automatisch erfolgen können. So ist es leicher möglich, während der Nutzung auftretende Fehler als Testfall zu implementieren, so dass die Fehler reproduzierbar und lösbar sind. Vor allem lässt sich so nachträglich jederzeit feststellen, dass durch Änderungen keine schon gelösten Fehler erneut auftreten.

7. Fazit

In der vorliegenden Arbeit wurde mit dem Ziel, eine bidirektionale und automatische Synchronisation zu ermöglichen, das Konzept eines synchronisierenden Peer-to-Peer Dateisystems entwickelt und anschließend dieses Konzept prototypisch implementiert.

Zunächst wurden Grundlagen der Synchronisation (Kapitel 2 (S. 3)) erläutert, um mögliche Probleme auzuzeigen. Außerdem wurde aufgezeigt, wie eine erfolgreiche Synchronisation erfolgen kann. In diesem Zusammenhang wurde ebenfalls aufgezeigt, wie Konflikte erkannt werden können. Zur Vereinfachung wurden die Grundlagen als Definitionen festgehalten.

Anschließend wurden Möglichkeiten der Synchronisation (Kapitel 3 (S. 22)) vorgestellt. Die vorgestellten Anwendungen decken einen großen Anwendungsbereich ab, von einfacher Synchronisation zu Datensicherungszwecken bis zur automatischen Synchronisation im professionellen Umfeld. Eine einfach zu nutzende automatische Synchronisation zwischen gleichberechtigten Computern existierte jedoch nicht.

Nachdem die Anforderungen an eine automatische Synchronisation definiert wurden, wurde eine Konzeption des synchronisierenden Peer-to-Peer Dateisystems (Kapitel 5 (S. 47)) entwickelt. Dieses Konzept stellt eine andere Art der Synchronisation dar, da die Synchronisation nur für einen einzelnen Anwender erfolgt, der seine Dokumente, Bilder oder digitialisierte Musik synchronisieren will. Entscheidend ist hierbei, dass keine zentrale Instanz notwendig ist, um die Synchronisation zu ermöglichen. Das kurzzeitige Betreiben der beteiligten Computer sollte ausreichen, die Dateien zu synchronisieren. Ein manuelles Starten der Synchronisation ist ebenfalls nicht notwendig.

Dieses Konzept wurde abschließend in einer Implementierung (Kapitel 6 (S. 75)) prototypisch umgesetzt. Dabei wurde in einigen Punkten von dem vorgestellten Konzept abgewichen, um Probleme im Betrieb zu vermeiden. Die hier vorgestellte Implementierung erlaubt lediglich die Synchronisation der Dateien, Konflikte lassen sich jedoch leider nicht eerkennung und lösen.

Es wurde gezeigt, dass es möglich ist, ein synchronisierendes Peer-to-Peer Dateisystem zu entwickeln. Auch wenn noch nicht alle gestellten Anforderungen umgesetzt

worden sind, kann diese prototypische Implementierung weiterentwickelt werden. Außerdem können durch die häufige oder alltägliche Nutzung weitere Fehler gefunden und Informationen gesammelt werden, um weitere Optimierungsmöglichkeiten zu erarbeiten und umzusetzen.

8. Ausblick

Das synchronisierende Dateisystem in seiner jetzigen, prototypischen Implementierung ist erweiterbar. Durch kontinuierliche Nutzung und Evaluierung können Informationen über die Nutzung im alltäglichen Einsatz gesammelt werden. Diese Informationen können anschließend als Ausgangspunkt für Verbesserungen herangezogen werden. Anhand der in der Testphase gesammelten Informationen werden, neben den schon genannten Optmierungsmöglichkeiten (siehe Kapitel 5.4.4 (S. 68) und Kapitel 6.3.1 (S. 102)), in diesem Kapitel weitere Verbesserungsmöglichkeiten erläutert.

Eine weitere Verbesserungsmöglichkeit besteht darin, die Netzwerkkommunikation auszulagern. Das Konzept geht davon aus, dass jedes Dateisystem die Kommunikation mit anderen synchronisierenden Dateisystemen selbstständig handhabt. Werden auf einem System jedoch mehrere synchronisierende Dateisysteme verwendet, bedeutet dies, dass jedes Dateisystem einen eigenen IP-Port verwendet. Werden zwei Dateisysteme beispielsweise für eine Datensicherung auf einem externen Datenträger verwendet, erfolgt die Kommunikation ebenfalls über TCP/IP. Zudem werden zwei IP-Ports benötigt. Wird die Netzwerkkommunikation ausgelagert, könnte ein *Daemon* diese Aufgabe übernehmen (vgl. Tanenbaum 2003*b*, S. 740). Für die Kommunikation mit dem jeweiligen Dateisystem könnte eine lokale Kommunikation, wie etwa *Unix Domain Sockets*, genutzt werden. *Unix Domain Sockets* sind der TCP/IP-Kommunikation sehr ähnlich, arbeiten jedoch nur lokal und können somit auf viele Sicherungsmechanismen verzichten, so dass diese Kommunikation effizienter erfolgen kann. Zudem würde so nur ein einziger IP-Port für eine externe Kommunikation verwendet werden müssen.

Alle genannten Anpassungen beziehungsweise Optimierungen verändern das Konzept nicht, sondern passen einige Komponenten nur etwas besser an die Verwendung durch den Anwender oder die Umgebung an.

Bisher wurde das synchronisierende Peer-to-Peer Dateisystem mit dem Ziel entwickelt, Dateien und Verzeichnisse eines Anwenders zwischen mehreren Computern oder Datenträgern zu synchronisieren. Im Folgenden sollen nun mögliche Erweiterungen

zur Nutzung durch mehrere Anwender betrachtet werden, die das Konzept in einigen Bereichen verändern.

Wenn mehrere Anwender zusammenarbeiten, tritt häufiger der Fall ein, dass diese untereinander Dateien austauschen. Verwaltet ein Anwender beispielsweise hilfreiche Dokumentation zu verschiedenen Themen, wäre es praktisch, wenn interessierte Kollegen diese „abonnieren“ könnten. So ließe sich das synchronisierende Dateisystem dahingehend erweitern, dass Anwender verschiedene Verzeichnisse für andere Anwender lesend freigeben. Diese könnten diese Verzeichnisse lesen und die entsprechenden Dateien synchronisieren. Dies erscheint zuächst klassischen Peer-to-Peer Netzwerken, wie beispielsweise *BitTorrent* (vgl. BitTorrent), sehr ähnlich zu sein. Es gibt jedoch entscheidende Unterschiede: Das synchronisierende Dateisystem synchronisiert aktiv, *BitTorrent* verteilt Dateien passiv. Aktualisiert der Anwender, der die Dokumentation verwaltet, einzelne Dokumente, so aktualisieren alle „Abonnenten“ ihre Kopien *automatisch*. Bei *BitTorrent* muss der Anwender selbstständig nach neuen Versionen suchen und diese dann „abonnieren“. In diesem Fall, ist das Hauptziel des synchronisierenden Dateisystems mehr eine unidirektionale und aktive Verteilung von Aktualisierungen.

Ein weiterer Schritt in diese Richtung ist, wenn mehrere Anwender sowohl lesend als auch schreibend Zugriff auf ein gemeinsames synchronisierendes Dateisystem haben. Mehrere Anwender greifen auf ein gemeinsames Dateisystem zu und bearbeiten die dort abgelegten Dateien. Jedoch liegen die Dateien auf den jeweiligen lokalen Datenträgern der Anwender und synchronisieren die Änderungen bei Bedarf. An diesem Punkt ist dann zusätzlich eine dezentrale Berechtigungsstruktur erforderlich. Dies könnte ermöglichen, dass es Anwender gibt, die Dokumente aktualisieren und Abonennten, die die Dokumente nur empfangen und nutzen, jedoch keine Änderungen vornehmen dürfen. Diese Möglichkeit würde die unidirektionale Verteilung und die bidirektionale Synchronisation in Abhängigkeit der Berechtigungen des Anwenders kombinieren.

Gegenüber anderen Peer-to-Peer Dateisystemen, wie das schon genannte *Ivy* (vgl. Muthitacharoen u. a. 2002) oder auch das *Freenet Project* (vgl. Freenet Project), werden bei dem hier vorgestellten synchronisierenden Peer-to-Peer Dateisystem die Dateien immer auf alle Synchronisationspartner verteilt. Jeder Synchronisationspartner enthält also alle abgelegten Dateien und Verzeichnisse; das Dateisystem formt keinen gemeinsamen großen Speicher, von dem jeder Teilnehmer einen Teil verwaltet. Es ist damit den *Distributed Version Control Systems*, wie *Mercurial* (vgl. Mercurial), *Bazaar* (vgl. Bazaar) oder *GIT* (vgl. GIT), sehr viel ähnlicher.

Diese verteilten Peer-to-Peer Versionskontrollsysteme sind eine relativ neue Entwicklung, alle zuvor genannten Systeme wurde im Jahr 2005 gestartet. Die zentralistischen Versionskontrollsysteme sind älter, die Arbeit an *CVS* (vgl. GNU CVS) begann etwa 1989 und *Subversion* (vgl. Collins-Sussman, Fitzpatrick und Pilato 2007, S. 2) wird seit dem Jahr 2000 entwickelt. Auch wenn mit *OpenCVS* (vgl. *Open*CVS) eine relativ neue Implementierung existiert, ist diese Implementierung dennoch kompatibel zu anderen und älteren Implementierungen. Ein wesentlicher Unterschied zu diesen Systemen besteht dennoch, da die Versionierung dieses synchronisierenden Peer-to-Peer Dateisystems nur der Synchronisation dient. Die Versionierung könnte im Rahmen einer Weiterentwicklung eingeschränkt oder sogar entfernt werden.

Durch die automatische Synchronisation sollte erreicht werden, dass der Anwender möglichst auf all seinen Computern mit den aktuellen Versionen seiner Dateien arbeiten kann. Durch das synchronisierende Dateisystem wurde dieses Ziel, wenn bisher auch prototypisch, erreicht. Bei Optimierungen und Erweiterungen sollte dieses Ziel erhalten bleiben.

A. Anhang

A.1. Inhalt der beigelegten CD-ROM

Dieser Arbeit liegt eine CD-ROM mit folgendem Inhalt bei:

`source` Verzeichnis	Quelltext der prototypischen Implementierung
`Gaerner_Masterarbeit.pdf`	PDF Version der Masterarbeit

Literatur

Alexandrescu, Andrei (2003). *Modernes C++ Design*. 1. Aufl. Bonn: mitp-Verlag. S.: 75, 77, 88, 92.

Alexandrescu, Andrei und Petru Marginean (2000). *Generic: Change the Way You Write Exception-Safe Code – Forever*. 01. Dez. 2000. URL: http://www.ddj.com/cpp/184403758 (besucht am 11. 11. 2008). S.: 78.

Balasubramaniam, S. und Benjamin C. Pierce (1998). „What is a file synchronizer?" In: *in Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98)*. S. 98–108. S.: 25, 26.

Beck, Kent (2003). *Extreme Programming. Das Manifest*. Addison-Wesley Verlag. ISBN: 3-8273-2139-5. S.: 104.

BitTorrent.org. *BitTorrent*. URL: http://www.bittorrent.org/ (besucht am 25. 11. 2008). S.: 109.

Bjureblad, Magnus. *Reflective Icons - Oxygen*. Lizenz GPL. URL: http://www.kde-look.org/content/show.php/?content=77300 (besucht am 09. 09. 2008). S.: 47.

Braam, Peter J. (1998). *The Coda Distributed File System*. http://www.cs.cmu.edu/afs/cs/project/coda-www/ResearchWebPages/docdir/lj98.pdf. URL: http://www.coda.cs.cmu.edu/ljpaper/lj.html (besucht am 27. 07. 2007). S.: 33, 34.

Canonical Ltd. and Bazaar development community. *Bazaar*. URL: http://bazaar-vcs.org/ (besucht am 23. 09. 2008). S.: 41, 109.

Carnegie Mellon University und IBM Pittsburgh Labs. *AFS*. URL: http://www.cs.cmu.edu/afs/andrew.cmu.edu/usr/shadow/www/afs.html (besucht am 27. 07. 2007). S.: 33.

Clarke, Ian, Matthew Toseland, Oskar Sandberg u. a. *The Free Network Project*. URL: http://freenetproject.org/ (besucht am 26. 11. 2008). S.: 109.

Collins-Sussman, Ben, Brian W. Fitzpatrick und Michael C. Pilato (2007). *Versionskontrolle mit Subversion*. 2. Aufl. O'Reilly Verlag GmbH & Co. KG. ISBN: 3-89721-460-1. S.: 27–29, 110.

Davison, Wayne, Andrew Tridgell und Paul Mackeras. *rsync*. URL: http://rsync.samba.org/ (besucht am 27. 07. 2007). S.: 23.

Developers, SQLite. *SQLite*. URL: http://www.sqlite.org/ (besucht am 28. 10. 2008). S.: 86, 87.

Free Software Foundation. *Concurrent Versioning System*. URL: http://de.wikipedia.org/wiki/Concurrent_Versions_System (besucht am 27. 07. 2007). S.: 9, 11, 27, 110.

FreeBSD Documentation Project (2008). *FreeBSD Handbook*. URL: ftp://ftp.freebsd.org/pub/FreeBSD/doc/en_US.ISO8859-1/books/handbook/book.pdf.bz2 (besucht am 09. 09. 2008). S.: 9.

FUSE Developers. *FUSE - Filesystem in Userspace*. URL: http://fuse.sourceforge.net/ (besucht am 27. 07. 2007). S.: 75, 81, 83.

Gamma, Erich u. a. (1996). *Entwurfsmuster*. 5. Aufl. München: Addison-Wesley Verlag. S.: 76, 80, 83, 88.

Guy, Richard G. u. a. (1990). „Implementation of the Ficus Replicated File System". In: *USENIX Conference Proceedings*. JSH: folder: Ficus publications: USENIX. S. 63–71. URL: ftp://ftp.cs.ucla.edu/pub/ficus/usenix_summer_90.ps.gz (besucht am 04. 09. 2008). S.: 37, 38.

Guy, Richard u. a. *Rumor: Mobile Data Access Through Optimistic Peer-to-Peer Replication*. URL: http://fmg-www.cs.ucla.edu/rumor98/er98wmda.pdf (besucht am 27. 02. 2008). S.: 37.

Handelsblatt (2008). *PC-Absatz wächst dank neuer Technik*. 02. Juli 2008. URL: http://www.handelsblatt.com/unternehmen/it-medien/pc-absatz-waechst-dank-neuer-technik;2007109 (besucht am 26. 11. 2008). S.: 1.

Heidemann, John S. und Gerald J. Popek (1991). *A Layered Approach to File System Development*. Techn. Ber. CSD-910007. University of California, Los Angeles. URL: ftp://ftp.cs.ucla.edu/pub/ficus/OLD_TECHREPORTS/ucla_csd_910007.ps.gz (besucht am 04. 09. 2008). S.: 37.

Heidemann, John S. u. a. (1992). „Primarily Disconnected Operation: Experiences with Ficus". In: *Proceedings of the Second Workshop on Management of Replicated Data*. IEEE. URL: ftp://ftp.cs.ucla.edu/pub/ficus/WorkMgtReplData_92.ps.gz (besucht am 04. 09. 2008). S.: 37, 38.

Hunt, Andrew und David Thomas (2003). *Der Pragmatische Programmierer*. 1. Aufl. München: Hanser Verlag. S.: 103.

IBM, Microsoft und Samba Team. *Server Message Block*. URL: http://de.wikipedia.org/wiki/Server_Message_Block (besucht am 29. 08. 2008). S.: 33.

Kernighan, Brian W. und Dennis M. Ritchie (1990). *Programmieren in C*. Carl Hanser Verlag. ISBN: 3-446-15497-3. S.: 83–85.

Kumar, P. und M. Satyanarayanan (1993). „Supporting Application-Specific Resolution in an Optimistically Replicated File System". In: *Proceedings of the Fourth IEEE Workshop on Workstation Operating Systems*. Napa, CA. S. 66–70. URL: http://www.cs.cmu.edu/afs/cs/project/coda-www/ResearchWebPages/docdir/wwos4.pdf (besucht am 22. 02. 2008). S.: 9, 10, 35.

Lehey, Greg (2006). *The Complete FreeBSD*. 4. Aufl. URL: http://www.lemis.com/grog/Documentation/CFBSD/ (besucht am 09. 09. 2008). S.: 57.

M. Satyanarayanan und Carnegie Mellon University Students. *Coda*. URL: http://www.coda.cs.cmu.edu/ (besucht am 27. 07. 2007). S.: 33.

McCutchan, John und Robert Love. *inotify*. URL: http://en.wikipedia.org/wiki/Inotify (besucht am 09. 01. 2008). S.: 49.

Mercurial Developers. *Mercurial*. URL: http://www.selenic.com/mercurial/wiki/ (besucht am 27. 08. 2008). S.: 30, 109.

Meyers, Scott (2006). *Effektiv C++ programmieren*. 3. Aufl. München: Addison-Wesley Verlag. S.: 75, 77, 91.

Microsoft, Refactored Networks, LLC und DataPower Technology, Inc. (2005). *A Universally Unique IDentifier (UUID) URN Namespace*. Juli 2005. URL: http://tools.ietf.org/html/rfc4122 (besucht am 04. 10. 2008). S.: 6, 65, 118.

Moolenar, Bram u. a. *VIM*. URL: http://www.vim.org/ (besucht am 10. 09. 2008). S.: 62.

Muthitacharoen, Athicha u. a. (2002). „Ivy: A Read/Write Peer-to-peer File System". In: *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*. Boston, Massachusetts. URL: http://pdos.csail.mit.edu/ivy/ (besucht am 26. 09. 2008). S.: 39, 109.

Open Mobile Alliance (2008). *A Primer to SyncML/OMA DS*. Techn. Ber. Open Mobile Alliance. URL: http://member.openmobilealliance.org/ftp/public_documents/ds/Permanent_documents/OMA-WP-SyncML_Primer-20080331-A.zip (besucht am 17. 04. 2008). S.: 2, 44.

OpenBSD Project. Open*SSH*. URL: http://www.openssh.org/ (besucht am 27. 07. 2007). S.: 71, 72, 83.

OpenBSD Project. *OpenCVS*. URL: http://www.opencvs.org/ (besucht am 25. 07. 2008). S.: 9, 11, 27, 110.

O'Sullivan, Bryan (2007). *Distributed revision control with Mercurial*. URL: http://hgbook.red-bean.com/ (besucht am 29. 08. 2008). S.: 30.

Pierce, Benjamin C. und Jérôme Vouillon (2004). *What's in Unison? A Formal Specification and Reference Implementation of a File Synchronizer*. Techn. Ber. MS-CIS-03-36. Dept. of Computer und Information Science, University of Pennsylvania. URL: http://www.cis.upenn.edu/~bcpierce/papers/unisonspec.pdf (besucht am 04. 09. 2008). S.: 1, 25.

Popek, Dr. Gerald und Dr. Peter Reiher (1996). *Ficus and Rumor*. URL: http://www.lasr.cs.ucla.edu/ficus/ (besucht am 29. 12. 2007). S.: 36.

Projekt, GnuPG. *libgcrypt*. URL: http://www.gnupg.org/related_software/libraries.en.html (besucht am 18. 11. 2008). S.: 97.

Rochkind, Marc J. (2004). *Advanced UNIX Programming*. 2. Aufl. Addison-Wesley. ISBN: 013-141154-3. S.: 88.

Satyanarayanan, M. u. a. (1993). „Experience with Disconnected Operation in a Mobile Computing Environment". In: *Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing*. Boston, MA. URL: http://www.cs.cmu.edu/afs/cs/project/coda-www/ResearchWebPages/docdir/mobile93.pdf (besucht am 16. 02. 2008). S.: 1, 33.

Schill, Alexander und Thomas Springer (2007). *Verteilte Systeme*. Spinger-Verlag Berlin Heidelberg. ISBN: 3-540-20568-3. S.: 30, 60.

Schiller, Jochen (2003). *Mobile Communications*. 2. Aufl. Pearson Education Limited. ISBN: 0-321-12381-6. S.: 2, 33, 37.

Schmeh, Klaus (2007). *Kryptografie. Verfahren, Protokoll, Infrastrukturen*. 3. Aufl. dpunkt.verlag GmbH. ISBN: 978-3-89864-435-8. S.: 8, 9, 70, 71, 97.

Schmidt, Douglas C. und Stephen D. Huston (2005). *C++ Network Programming – Mastering Complexity with ACE and Patterns*. Bd. 1. C++ In-Depth Series. [letzter Zugriff: 17. April 2008]. Addison-Wesley. ISBN: 0-201-60464-7. URL: http://www.riverace.com/. S.: 76.

– (2006). *C++ Network Programming – Systematic Reuse with ACE and Frameworks*. Bd. 2. C++ In-Depth Series. Addison-Wesley. ISBN: 0-201-79525-6. URL: http://www.riverace.com/ (besucht am 17. 04. 2008). S.: 76.

Schmidt, Douglas u. a. (2004). *Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects*. Bd. 2. John Wiley & Sons, Ltd. ISBN: 0-471-60695-2. S.: 75, 78, 96.

Sedgewick, Robert (1988). *Algorithms*. 2. Aufl. Addison-Wesley Publishing Company. ISBN: 0-201-06673-4. S.: 10.

Stallman, Richard M. und the GNU Project. *GNU Emacs*. URL: http://www.gnu.org/software/emacs/ (besucht am 10. 09. 2008). S.: 62.

Subversion Development Team. *Subversion*. URL: http://subversion.tigris.org/ (besucht am 27. 07. 2007). S.: 9, 27.

Sun Microsystems. *Network File System*. URL: http://de.wikipedia.org/wiki/Network_File_System (besucht am 27. 07. 2007). S.: 33, 34, 39.

Sutter, Herb und Andrei Alexandrescu (2005). *C++ Coding Standards*. 1. Aufl. New Jersey: Pearson Education. S.: 62, 75, 77, 85, 86.

Szeredi, Miklos. *SSH Filesystem*. URL: http://fuse.sourceforge.net/sshfs.html (besucht am 27. 11. 2007). S.: 83.

Tanenbaum, Andrew S. (2003*a*). *Computernetzwerke*. 4. Aufl. Pearson Studium. ISBN: 3-8273-7046-9. S.: 70–73, 96.

– (2003*b*). *Moderne Betriebssysteme*. 2. Aufl. München: Pearson Studium. ISBN: 3-8273-7019-1. S.: 6, 34, 56, 57, 61, 70, 84, 87, 93, 94, 108.

Tanenbaum, Andrew S. und Marten van Steen (2003). *Verteilte Systeme*. 1. Aufl. Pearson Studium. ISBN: 3-8273-7057-4. URL: http://www.cs.vu.nl/~ast/books/ds1/ (besucht am 29. 08. 2008). S.: 33.

– (2008). *Verteilte Systeme. Prinzipien und Paradigmen*. 2. Aufl. Pearson Studium. ISBN: 978-3-8273-7293-2. S.: 5, 8, 9, 13, 16, 17, 30, 32, 34, 35, 39, 57, 60, 63, 65, 70, 88.

Tanenbaum, Andrew S. und Albert S. Woodhull (1997). *Operating Systems. Design and Implementation*. 2. Aufl. Prentice-Hall Inc. ISBN: 0-13-638677-6. S.: 34, 81, 97.

Torvalds, Linus und Junio C Hamano. *GIT*. URL: http://git.or.cz/ (besucht am 23. 09. 2008). S.: 41, 109.

Tridgell, Andrew (1998). *The rsync algorithm*. Techn. Ber. URL: http://rsync.samba.org/tech_report/ (besucht am 09. 09. 2008). S.: 24.

– (1999). „Efficient Algorithms for Sorting and Synchronization“. Diss. The Australian National University. URL: http://samba.org/~tridge/phd_thesis.pdf. S.: 9, 17, 23, 24, 26.

Ts'o, Theodore u. a. *ext2 Dateisystem Tools*. URL: http://e2fsprogs.sourceforge.net/ext2.html (besucht am 24. 11. 2008). S.: 78.

Unison Development Team. *Unison*. URL: http://www.cis.upenn.edu/~bcpierce/unison/ (besucht am 27. 07. 2007). S.: 25.

Vesperman, Jennifer (2004). *CVS*. 1. Aufl. O'Reilly Verlag. ISBN: 3-89721-369-9. S.: 11, 27, 29.

Glossar

Daemon Bei unix-artigen Systemen werden Anwendungen, die ohne interaktive Schnittstelle zum Anwender im Hintergrund ausgeführt werden, als *Daemon* bezeichnet. Dies sind beispielsweise Anwendungen, die E-Mails versenden oder Druckaufträge abarbeiten., Seite 106

LRU-Cache Ein LRU-Cache verwaltet eine gewisse oder definierbare Anzahl von Objekten so, dass immer die zuletzt nachgefragten Objekte erhalten bleiben. Alle anderen Objekte werden nach und nach entfernt. Ein bekanntes Beispiel für den verwendeten Algorithmus ist die Liste der zuletzt geöffneten Dateien, die verschiedene Anwendungen bereitstellen., Seite 33

NAS Ein NAS ist einfach betrachtet eine Festplatte mit Netzwerkanschluss. Häufig lassen sich diese Geräte um zusätzliche Festplatten erweitern. Zudem unterstützen sie meistens mehrere Protokolle, um auf die Dateien zuzugreifen wie beispielsweise *SMB* und *FTP (File Transfer Protocol)*. Ebenso ist häufig eine Benutzerverwaltung integriert, um den Zugriff auf die Dateien zu verwalten., Seite 42

SyncML *SyncML* definiert eine Architektur und Vorgehensweise, um Daten zu synchronisieren und Konflikte zu erkennen. Viele Mobilfunktelefone verwenden diesen Standard, um ihre Daten wie Adressbücher mit anderen Computern abzugleichen., Seite 2

UUID *UUID* oder *GUID* steht für **U**niversally **U**nique **Id**entifier) beziehungsweise **G**lobally **U**nique **Id**entifier) und dient dazu, dezentral eindeutige Kennungen zu erzeugen. *RFC 4122 (Request for Comment)* (vgl. Microsoft, Refactored Networks, LLC und DataPower Technology, Inc. 2005) spezifiziert die Generierung dieser eindeutigen Kennungen. Eine *UUID* ist 128 Bit lang und wurde ursprünglich im *Apollo Network Computing System*, später auch im *Distributed Computing Environment DCE* der *Open Software Foundation OSF* und auch unter *Microsoft Windows* genutzt., Seite 6

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung des Masterstudiengangs Informatik an der Hochschule für Angewandte Wissenschaften Hamburg nach §22(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 28. November 2008

Ort, Datum

Unterschrift