Seminar Langzeitarchivierung Versionsverwaltung in der Dateisystemebene

Arthur Müller
1071312
14.05.2009

Aufgabensteller:
Prof. Dr. Uwe M. Borghoff
Betreuer:
Dipl.-Inform. Sebastian Rönnau

Institut für Softwaretechnologie
Fakultät für Informatik
Universität der Bundeswehr München

Inhaltsangabe

Im Rahmen des Seminars zur Langzeitarchivierung befasst sich diese Arbeit mit der Versionierung der Daten in der Dateisystemebene. Diese Arbeit beschäftigt sich mit den bekannten Verfahren und deren Eigenschaften, dabei v.a. mit der Transparenz. In wie weit sich solche Dateisysteme zum direkten oder indirekten Einsatz in Bezug auf Langzeitarchivierung eignen, ist ein weiterer bedeutender Aspekt dieser Arbeit.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

Die Begriffe der Langzeitarchivierung, Datensicherung und Versionsverwaltung hängen stark miteinander zusammen. Auch die naivste Art der Datensicherung wie z.B. vollständige Backup werden meistens nach einem Datum abgelegt, das die Rolle der Versionsmarke übernimmt.

Versionsverwaltung kann auf unterschiedlichsten Ebenen stattfinden. Angefangen mit Hardware, die im Hintergrund die Datensicherung und Versionierung übernimmt, und aufgehört bei Userland-Tools mit dem Repository-Konzept wie das ältere `RCS` [1], `CVS` [2] und das neue `SVN`, die v.a. auf die Versionierung des Quellcodes ausgelegt sind. Das Konzept der Versionierung in die Dateisystemebene zu verlegen, zielt darauf ab kostspielige Hardware-Implementierungen zu vermeiden und gleichzeitig im Vergleich zu den bereits genannten Userland-Programmen mehr Transparenz für den Benutzer zu erreichen und Versionierung auf alle Dateitypen zu verallgemeinern (Abbildung 1).

Diese Arbeit beschäftigt sich mit den Verfahren zur Versionsverwaltung in der Dateisystemebene, den späteren Zugriffsmöglichkeiten auf gespeicherte Versionen, v.a. im Bezug auf Langzeitarchivierung, der Transparenz und der Datensicherheit solcher Dateisysteme.

2 Verfahren zur Versionsverwaltung

Die einfachste Art eine neue Version zu erstellen ist es, das Original zu kopieren, mit einem Versionsvermerk zu ergänzen, und dann weiterhin mit dem Original zu arbeiten. Dieses Verfahren ist als Vollsicherung bekannt und kann große Mengen des Speichermediums verbrauchen. Weiterhin existieren auch differentielle und inkrementelle Verfahren, die nur noch auf die Änderungen der Daten eingehen und diese sichern bzw. als ältere Version speichern [3].

2.1 Explizite Versionierung

Frühere Dateisysteme unterstützten eine Dateiversionierung über spezielle Tools bzw. Kommandos [4][5]. Solche Befehle wie `checkin`, `checkout` oder `commit` erinnern stark an die Userland-Tools wie `CVS`, die sich in der Applikationsebene der Versionierung befinden (Abbildung 1). Auch eine Kennzeichnung bestimmter Versionen ist in manchen Fällen möglich [6]. Der größte Nachteil dieser Semantik besteht darin, dass sie nicht transparent gegenüber dem Benutzer ist. Entsprechende Kommandos müssen erlernt werden, was unter Umständen viel Zeit in Anspruch nimmt. Die Versionierung bedarf der

Kooperation des Anwenders. Dieser entscheidet selbst über die Notwendigkeit der Versionierung für Dateien. Eine Kontrolle darüber von der höheren Instanz ist nicht umsetzbar.

2.2 Snapshots

Snapshots sind im Allgemeinen nur mit Leserechten beschränkte Kopien des gesamten Dateisystems, also aller sich darauf befindlichen Dateien. Snapshots werden im Wesentlichen dadurch charakterisiert, dass diese mit differentiellen oder inkrementellen Methoden zur Sicherung bzw. Versionsverwaltung verbunden werden. Ein Snapshot ohne solche Erweiterung würde der Vollsicherung gleichen und enorme Mengen an Speicherplatz verbrauchen.

Die Snapshots können verschieden initialisiert werden und zu unterschiedlichen Zeitpunkten geschehen. Die Durchführung der Snapshots kann direkt vom Dateisystem in gewissen Intervallen übernommen werden [7]. Sind die Intervalllängen fest und können nicht verändert werden, wie stündlich oder täglich, so kann keine vollständige Aussage darüber getroffen werden, welche Aktionen auf dem Dateisystem in der Zwischenzeit ausgeführt wurden. Temporäre Ablage illegaler Daten mit der kürzeren Existenzzeit als die Intervalllänge kann hierbei nicht verhindert werden. Eine Entschärfung bietet die Möglichkeit, die Intervalle unregelmäßig zu gestalten, so dass der Versuch das System zu umgehen, erschwert wird. Auch die Verkürzung der Intervalle kann das Problem nicht komplett lösen und bringt zusätzlich viel unnötiges Overhead in den Systemleerlauf. Sind Konfigurationsmöglichkeiten geboten, so erfolgen die Einstellungen über spezielle Tools. Mitgelieferte Programme könnten über `cron`-Jobs im Hintergrund gestartet werden [8]. Der Administrator hat dann zwar mehr Einfluss, aber das System verliert an Transparenz. Ein weiteres gewichtiges Problem besteht darin, dass Ände-

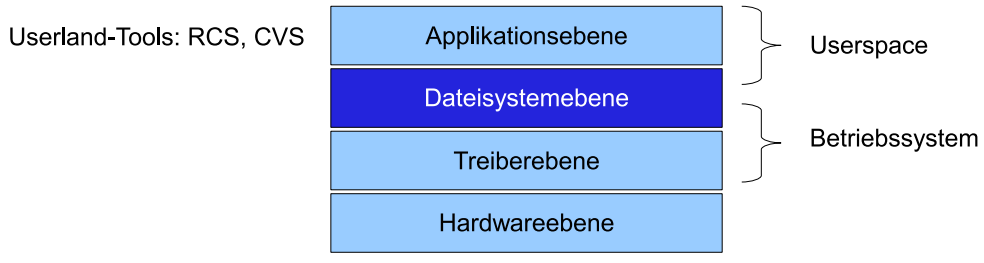

Abbildung 1: Ebenen der Versionsverwaltung

rungen zwischen zwei Snapshots nicht wiederherstellbar sind [9]. Es ist nur möglich zu einem älteren Stand zurückzukehren. Unterschiede zwischen zwei aufeinanderfolgenden Snapshots sind nicht ohne weiteres erfassbar. Auch werden alle Dateien gleich behandelt, obwohl nicht jede Datei gleich häufig benutzt wird, was mit unnötigem Speicherplatzverbrauch und Overhead verbunden ist. Beschränkung auf bestimmte Dateien oder Verzeichnisse kann ebenfalls nur mit speziellen Tools oder Konfigurationen erfolgen. Höhere Effizienz der Speicherplatznutzung geht auf Kosten der Transparenz. Eine weitere Möglichkeit ein Snapshot zu erstellen, die absolute Transparenz bietet, wird mit dem Begriff "On Demand" ausgezeichnet, d.h. die Versionen werden bei bestimmten Aktionen so erstellt, dass der Nutzer nichts davon bemerkt, z.B. beim Abspeichern einer Datei. Solche Vorgehensweise wird meist in Verbindung mit Copy-on-Write-Techniken realisiert.

2.3 Copy-on-Write

Im Prinzip handelt es sich hierbei um eine Optimierungstechnik, die hauptsächlich darauf ausgelegt ist Speicherplatz einzusparen. Wenn zum Beispiel verschiedene Aufrufer (caller) den gleichen Speicherabschnitt benötigen, so kann man diesen den gleichen Zeiger auf den gegebenen Abschnitt zuweisen und als solches nur einmal im Speicher bereit zu halten, solange keiner der Aufrufer diesen verändern will. Wird eine Veränderung von einem der Aufrufer verlangt, z.B. durch ein `write`, so wird für den Aufrufer eine private Kopie erstellt, auf der dieser weiterarbeiten kann [10] (Abbildung 2).

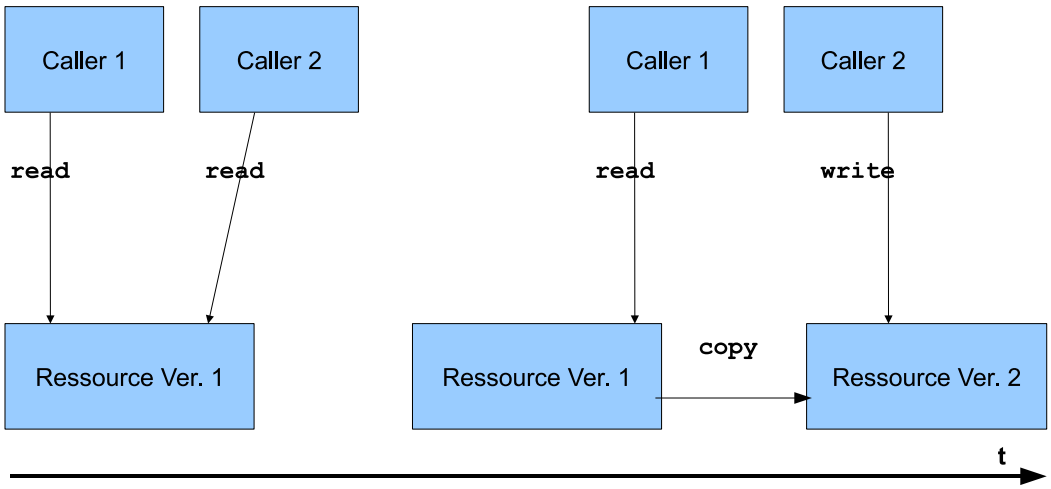

Abbildung 2: Darstellung von Copy-on-Write im Allgemeinen

In den Dateisystemen gewinnt diese Technik andere Eigenschaften. Wird

also eine Datei neubeschrieben, so entsteht eine neue Kopie, die als aktuelle Version agieren kann. Das alte Original wird dann zu einer Vorversion. Die Kopien können unterschiedlichst gestaltet sein: vollständige, inkrementelle oder differentielle. Vollständige Kopien verbrauchen viel Platz, beschleunigen aber den Zugriff auf ältere Versionen der Dateien [8]. Inkrementelle und differentielle Kopien haben die umgekehrten Eigenschaften. Versionierungsdateisysteme bieten unterterschiedliche Modi an: `full`, `sparse`, `compressed` etc. Bei dem `sparse mode` handelt es sich um eine inkrementelle Technik, die Deltas auf ganze Blöcke ausweitet, d.h. ändert sich etwas in einem Block, so wird dieser komplett neu gespeichert [11], . Copy-on-Write muss sich nicht nur auf Dateien beschränken. Auch der Einsatz beim Journaling ist sinnvoll und verbessert die Sicherheit der Konsistenzwiederherstellung nach einem Systemabsturz [12].

2.3.1 Copy-on-Change

Diese Technik ist eine Abwandlung von Copy-on-Write. Bei Copy-on-Write wird im Allgemeinen immer dann eine neue Version angelegt, wenn der Systemaufruf `write` kommt. Copy-on-Change wurde als getrennter Begriff von Muniswamy-Reddy [11] verwendet. In diesem Fall wird nur dann eine neue Version geschrieben, wenn die Daten sich tatsächlich geändert haben. Das System überprüft also die aktuellen und die neuen Daten Byte-für-Byte auf Unterschied. Das bringt zusätzliches Overhead bei jedem Schreibzugriff, verringert aber Speicherplatzverbrauch.

3 Versionierung und weitere Langzeitarchvierung

3.1 Zugriffsrechte

Nicht alle Versionierungsdateisysteme bieten einen direkten on-line Zugriff auf die älteren Versionen [13][14]. Interne Versionsverwaltung mit Snapshots wird oft nur dafür genutzt, um einen Rollback zu dem letzten konsistenten Zustand nach einem Systemabsturz zu ermöglichen. Dies gleicht einem Recovery nach außen; mit dem Unterschied, dass die Dateisysteme trotzdem mehrere Versionen bereit halten, um die Sicherheit eines sauberen Rollbacks zu erhöhen. Solche Dateisysteme eignen sich schlecht für die Langzeitarchivierung, da nur aktuelle Versionen dem Nutzer zur Verfügung stehen und die Versionierung zur Abspeicherung wieder von anderen Systemen übernommen werden müssen.

3.1.1 Überlauf des Dateisystems durch den Versionsoverhead

In manchen Dateisystemen ist nur ein Lesezugriff auf die versionierte Dateien, z.B. über Snapshots, möglich [8]. Wurde ein Snapshot gemacht, so kann dieser nicht mehr entfernt werden. Der Speicher ist zwar nach einer gewissen Zeit voll, allerdings kann auch eine solche Methode auch durchaus sinnvoll sein. Wenn ein Unternehmen verhindern möchte, dass auf den Computern der Mitarbeiter illegale Daten abgespeichert werden, kann ein solches Dateisystem zum Einsatz kommen. Anhand der älteren Versionen können Daten ausgelesen werden, die dem Benutzer gelöscht erscheinen und die derjenige verbergen wollte. Das große Problem tritt auf, wenn die Platte oder ein anderes Speichermedium voll ist. Die Daten müssen dann auf Bänder oder andere Medien, die sich zur Langzeitarchivierung eignen, übertragen werden und die aktuellen Versionen zwischengespeichert werden, damit diese nach der Neuformatierung der Festplatte dem Nutzer wieder verfügbar gemacht werden können. Spiegelt man die gesamte Festplatte auf den Band, so überträgt man eventuell zusätzlich das gesamte Betriebssystem und verschwendet damit Speicherplatz. Um dies zu verhindern, müssen bereits im Vorfeld Speichermedien sinnvoll eingebunden werden. Dabei sollte das Betriebssystem von der Datenpartition getrennt und mit Leserechten für den Benutzer eingeschränkt sein. Die beste Lösung zur Langzeitarchivierung wäre es, solche Dateisystemtypen direkt auf dem Archivmedium einzusetzen. Allerdings sind die meisten Dateisysteme für die Nutzung auf den Festplatten oder Medien mit nichtlinearem Zugriff auf die Daten ausgelegt und können aus diesem Grund nur kaum für Archivmedien verwendet werden.

Löschen der älteren Versionen durch das System im Hintergrund löst das oben beschriebene Problem [6][7]. Die Löschung der Vorversionen kann nach vordefinierten oder eingestellten Kriterien erfolgen, z.B. beschränkt durch die Anzahl der Versionen, die Zeit oder Platzverbrauch. Diese Einstellungen können sowie vom Nutzer als auch vom Administrator vorgenommen werden, wobei diejenigen vom Administrator in manchen Fällen die Priorität haben, so z. B. wenn ein Administrator zusichern will, dass die Mindestzahl der Versionen gewisse Zahl übersteigt [11]. Auch hier tritt das leidige Thema der verschlechterten Transparenz auf Kosten der Sicherheit auf.

3.1.2 Zugriff auf alte Versionen beim weiteren Archivieren

Wenn das Dateisystem die alte Versionen im Hintergrund bereinigt, diese aber zusätzlich auf einem externen Archivmedium gesichert werden müssen, dann stösst man auf das Problem der Abstimmung der dateisysteminternen Vorgänge und der nebenläufigen Prozesse, die die Sicherung übernehmen. Eventuell könnte das Dateisystem eine Schnittstelle bieten, um beide Prozeduren zu verbinden. Dabei übernimmt das Dateisystem die Initiative

zum Anstoßen des Kopierprozesses und der Administrator muss lediglich das passende Programm liefern. Allerdings muss das Dateisystem in diesem Fall wissen, in welcher Form die Daten benötigt werden. Liefert das Dateisystem nur Dateistücke, die intern mit inkrementellen Techniken gespeichert sind, so dass immer die Nachfolgeversion zur Wiederherstellung benötigt werden, so ist die Einholung der letzten archivierten Version auf dem externen Medium nur in Verbindung mit der aktuellsten Version möglich. Diese befindet sich auf der Festplatte. Wird die Fesplatte beschädigt, ist auch das gesamte Archiv nutzlos. Wenn die Daten von der Vorgängerversion aus berechnet werden, so leidet die Lesegeschwindigkeit darunter, vor allem bei den Bändern, die ohnehin langsam arbeiten. Um die aktuelle Version zu berechnen, müssen dann alle Teilstücke linear zusammengesetzt werden. Theoretisch darf das Dateisystem, dann auch nichts löschen, denn man braucht alle Teilstücke vom Anfang, es sei denn, vor dem Löschen der ältesten Version wird die Nachfolgeversion neuberechnet, so dass diese dann als älteste auf der Platte agieren kann [15].

Dateisysteme im Userspace [16] stellen zwar keine Möglichkeit zum Löschen der alten Dateien über die Schnittstelle, die der Benutzer kennt, dar, können diese andererseits auch nicht schützen. Solche Dateisysteme sind stark benutzerorientiert ausgelegt und betreiben Versionierung v.a. gegen eine zufällige Löschung [17]. Andererseits zwingen Dateisysteme zu keinen Umwegen, um die Daten auf einem Langzeitmedium zu archivieren. Die `raw`-Daten könnten mit gängigen Befehlen aus dem alten Verzeichnis einfach herauskopiert werden, allerdings auch nur dann, wenn die Metadaten zu den Dateien nicht zentral verwaltet werden, sondern jeder Version einzeln beiliegen.

3.2 Dezentrales Verwalten der Metadaten zur effizienten Speicherausnutzung

Jedes Dateisystem mit Versionierungsfähigkeiten versucht mehrere Dateien als eine einzige darzustellen. Natürlich bekommt der Benutzer immer die aktuelle Version zu sehen und die älteren Daten werden im Hintergrund verwaltet. Hierfür werden Metadaten verwendet, d.h. zusätzliche Daten, die das Dateisystem braucht, um mehrere Versionen in einer Sammlung zusammenzufassen. Solche Zusatzdaten könnten die Versionsnummern, Verweise auf Blöcke mit älteren Daten etc sein. Es gibt zwei wesentlich unterschiedliche Möglichkeiten Metadaten zu speichern: in Inodes an sich bzw. in einem zusätzlichen Block, auf den ein Inode verweist, oder in separaten Dateien mittels Dateisystemaufrufe.

3.2.1 In parallelen Dateien

Zadok nennt in seiner Arbeit den Begriff der parallelen Dateien (parallel files), um in diesen entweder ältere Versionen bei kompletten Kopien oder Metadaten bei differentiellen oder inkrementellen Kopien zu halten [18]. Bei der Speicherung der Vorversionen in parallelen Dateien unter Nutzung der vollständigen Kopien beschränken sich die Metadaten auf die Versionsnummern in Dateinamen. Werden inkrementelle oder differentielle Techniken verwendet so sind Metadaten zur Wiederherstellung erforderlich. Werden diese zentral pro Versionensammlung in einer separaten Datei z.B. als Logs [17] verwaltet, so erschwert dies die Langzeitarchivierung. Dateien mit Metadaten könnten von einem weiteren Versionierungssystem erfasst werden. Dieses System muss den Aufbau der Metadaten kennen, damit es den Versionen zugeordnet werden kann. Um die Versionierung der Metadaten zu vermeiden, kann auch die Metadatendatei bei jedem Archivierungszyklus auf dem Archivmedium überschrieben werden. Da solche Medien wie Bänder allerdings die Daten linear ablegen, ist das Überschreiben der alten Metadatendatei nicht möglich. Eine erneute Ablage der Metadatendatei bei jedem Zyklus führt im Extremfall dazu, dass mit jeder neuen Version auch neue Metadaten dazukommen, wobei die meisten sich bereits wiederholen. Die Hälfte des Bandes wäre nur mit solchen Dateien belegt und eine Platzverschwendung dieses Ausmaßes nicht akzeptabel. Aus diesen Gründen ist die Verwaltung der Metadaten in zentralen Dateien in Bezug auf Langzeitarchivierung nicht empfehlenswert.

3.2.2 In den Inodes oder zusätzlichen Blöcken

Durch die Verlagerung der Metadaten über Versionen in die Inodes, können diese komplett unsichtbar einem Nutzer gegenüber gemacht werden. Dabei wird auch höhere Geschwindigkeit beim Lesen erreicht, denn das Dateisystem muss keine zusätzliche Datei öffnen, um eine Version zu berechnen [6][8] (Abbilgung 3). Sind die erforderlichen Metadaten zu umfangreich, um diese direkt in den Inodes als erweiterte Attribute einzubauen, so kann auch durch ein Inode-Feld auf einen zusätzlichen Block verweisen werden, der praktisch am Ende jeder Datei angehängt wird [11]. Das Dilemma der zentralen Metadaten würde bei dieser Lösung nicht auftreten, denn die Metadaten wie gefodert jeweils der entsprechenden Version beiliegen. Allerdings können die Metadaten über gängige Dateisystemaufrufe auch gar nicht eingeholt werden, denn diese sind im spezifischen Inode versteckt. Nur durch spezielle Tools können diese geholt werden. Für das Ablegen dieser Metadaten würden zwei Optionen in Frage kommen: ein eigenes Verwaltungswerkzeug aufbauen; oder das gleiche Dateisystem nutzen, so dass die Inode-Struktur eins zu eins übertragen werden kann. Das Zweite allerdings schränkt die Flexibilität

enorm ein.

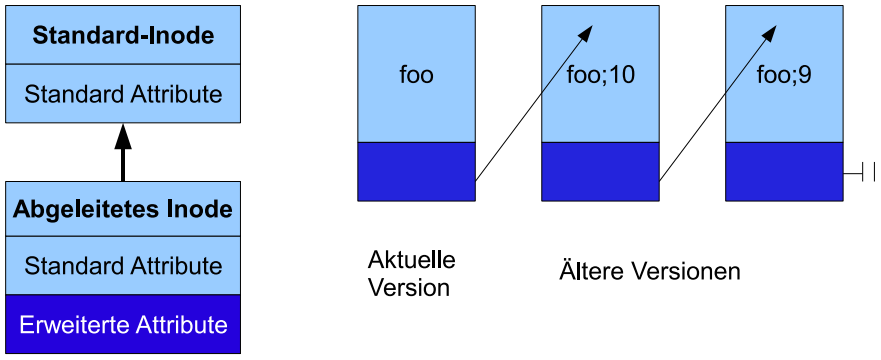

Abbildung 3: Verwaltung der Versionen mittels Metadaten in den Inodes

3.3 Darstellung der Versionen

Die wohl bekannteste Art auf Versionen zuzugreifen basiert auf den Versionsnummern. Diese können aus einer oder mehreren Zahlen zusammengesetzt sein, wie es auch bei den üblichen Software-Releases der Fall ist, z.B. Linux-Kernel 2.6.23. Manche der Dateisysteme machen sich diese Art der Versionierung zunutze und nummerieren die ältere Versionen mit einer fortlaufenden Zahl, wobei die aktuelle Version gar keine Nummer besitzt [7][11][17], was der Versionsvergabe in den Userland-Tools wie `RCS` oder `CVS` ähnelt. Möchte man zu einem Stand in Vergangenheit zurückkehren und man kennt man die Versionsnummer nicht, so wird man gezwungen sein, mehrere ältere Versionen einzuholen. Je weiter die Version in der Vergangenheit liegt, desto größer ist die Wahrscheinlichkeit, eine falsche Version beim Wiederherstellen vorzufinden. Bei den Archivmedien, die ohne hin langsam sind, kann dies beträchtliche Zeitmengen rauben. Um dieses Problem zu lösen, kann der Verlauf der Versionierung als eine kontinuierliche Zeitgerade und einzelne Versionen als Zeitpunkte (points of time) betrachtet werden (Abbildung 4). Der Zugriff auf die Versionen geschieht dann über die Angabe eines Zeitstempels (timestamp) [6][8]. Die Vorstellung der kontinuierlichen Zeitgerade ist natürlich nur eine Idealvorstellung. Die tatsächlichen Versionen werden diskret abgelegt und z.B. mit einer Epoch-Nummer versehen. Beim Aufruf der Datei zu einem gegebenen Zeitpunkt wird die nächst ältere Version genommen. Auch ein Zugriff über die bei der Snapshoterstellung vergebene Namen, sogenannte "named snapshots", ist hierbei möglich [8][9]. Über den Namen kann die Version am genauesten bestimmt werden. Die Transpa-

renz leidet wiederum darunter, da vorausgesetzt wird, dass der Benutzer die Snapshots manuell erstellt hat.

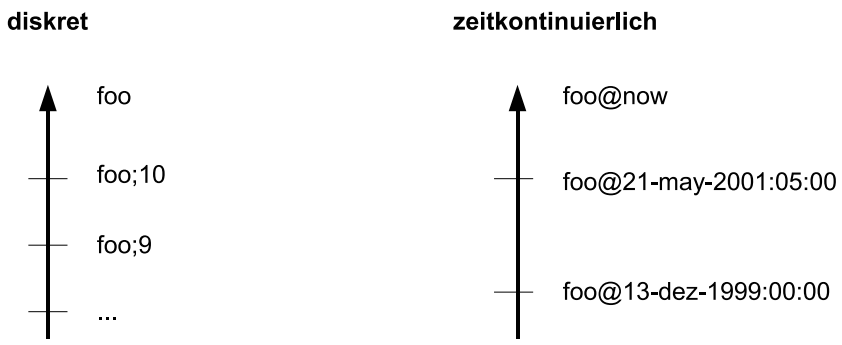

Abbildung 4: Diskrete und zeitkontinuierliche Darstellung der Versionen

4 Zusammenfassung

Im Allgemeinen können also zwei grundsätzlich unterschiedliche Dateisystemtypen mit den Versionierungsfähigkeiten beschribeben werden: Recovery- und Benutzer-orientierte. Natürlich sind es nur zwei Extrema und die meisten Dateisysteme befinden sich im Raum dazwischen. Recovery-orientierte Dateisysteme bieten meist komplette Transparenz gegenüber dem Benutzer. Diesen hat es überhaupt nicht zu kümmern, was intern passiert [13]. Andererseits hat der Nutzer im Extremfall keine Möglichkeit auf die Vorversionen zuzugreifen. Solche Dateisysteme sind darauf ausgelegt, nach einem Systemabsturz einen sauberen Rollback zu dem letzten konsistenten Zustand zu ermöglichen. Interne Versionierung hingegen wird nur als Mittel zum Zweck genutzt. Benutzer-orientierte Dateisysteme agieren im Extremfall genau umgekehrt und bieten dem Nutzer alle Rechte und Einblicke in die interne Versionierung [17]. Die vollständige Transparenz und gleichzeitig viele Rechte in einem System einzubauen, ist ohne entsprechende Anpassung der Betriebssystemkonzepte kaum möglich [18][19][20][21], dennoch kann man sich mit den beschriebenen Konzepten und Verfahren sich an diese Vorgaben nähern. Snapshots eignen sich gut für Backup- und allgemein Benutzer-orientierte Dateisysteme, in denen keine Zusicherung über den Stand zwischen zwei Snapshots getroffen werden muss. Mit dem Copy-on-Write-Prinzip wird optimale Transparenz für einen Benutzer beim Schreibvorgang erreicht werden, gleichzeitig erreicht man auch die Atomisierung, die bei den Snapshots nicht möglich ist. Die explizite Versionierung bietet weder Transparenz noch Si-

cherheit und ist aus diesen Gründen nicht ausreichend. Dezentralisierung der Metadaten und zeitlich basierte Darstellung der Versionen sind in Bezug auf Langzeitarchivierung wichtig und ersparen viel Overhead.

Die Dateisysteme mit Versionierungsfähigkeiten sind v.a. auf Festplatten oder Medien mit dem nichtlinearen Zugriff ausgelegt. Deswegen können diese nicht ohne weiteres direkt als Konzept für die Langzeitarchivierung übernommen werden. Wie bereits in früheren Kapiteln verdeutlicht, stösst man auch bei der indirekten Nutzung auf viele weitere Probleme. Derartige Konzepte verlangen mehr Sicherheit in Bezug auf die Löschung der Daten vom Benutzer, gleichzeitig aber freie Einsicht für den Benutzer und mehr Flexibilität für den Administrator.

Literatur

[1] H. Herold, M. Meyer: *SCCS und RCS, Versionsverwaltung unter UNIX*, Addison-Wesley, Bonn, Paris, 1995.

[2] D. Thomas, A. Hunt: *Versionsverwaltung mit CVS*, Carl Hanser Verlag, München, Wien, 2004.

[3] W. Barth: *Datensicherung unter Linux*, Open Source Press, München, 2004, Kapitel 2.

[4] D. K. Gifford, R. M. Needham, M. D. Schroeder: *The Cedar File System*: Communication of the ACM, 1988, S.288–298.

[5] D. G. Korn und E. Krell: *The 3-D File System*: In Proceedings of the USENIX Summer Conference, Sommer 1989, S.147–156.

[6] D. S. Santry et al.: *Deciding when to forget in the Elephant file system*, 17th ACM Symposium on Operating Systems Principles, San Francisco,

[7] D. Hitz, J. Lau, M. Malcolm: *File System Design for an NFS File Server Appliance*, The USENIX Association, San Francisco, Winter 1994.

[8] Z. N. J. Peterson, R. Burns: *Ext3cow: A Time-Shifting File System for Regulatory Compliance*, Vol. 1, No. 2, ACM Transactions on Storage, New York, Mai 2005.

[9] R. Dietze, T. Heuser, J. Schilling: *OpenSolaris für Anwender, Administratoren und Rechenzentren*, Springer Verlag, Berlin, Heidelberg, 2006,

[10] R. Brause: *Betriebssysteme, Grundlagen und Konzepte*, 2., überarbeitete Auflage, Springer Verlag, Berlin, Heidelberg, 2001, S.146, 159.

[11] K. Muniswamy-Reddy: *VersionFS: A Versatile and User-Oriented Versioning File System*, Stony Brook University, New York, Dezember 2004.

[12] Z. Brown: *NEXT-GEN FILESYSTEM TOPICS, BTRFS*: 2008 Linux Storage & Filesystem Workshop (LSF '08), San Jose, CA, Februar 2008.

[13] P. Padala: *A Log Structered File System with Snapshots*, University of Michigan, Juni 2005.

[14] J. E. Johnson, W. A. Laing: *Overview of the Spiralog File System*: In, Digital Technical Journal, Vol. 8 No. 2, 1996, S.5–14.

[15] H. Garcia-Molina: *Database Systems: The Complete Book*, Prentice Hall, München, 2002, Kapitel 17.

[16] M. Szeredi: *Filesystem in Userspace*, http://sourceforge.net/projects/avf.

[17] B. Cornell, P. A. Dinda, F. E. Bustamante: *Wayback: A User-level Versioning File System for Linux*, Computer Science Department, Northwestern University, Evanston/Chicago, 2004.

[18] E. Zadok et al.: *On Incremental Filesystems*, Vol. 2 No. 2, ACM Transactions on Storage, New York, Mai 2006.

[19] S. Müller und S. Widmer: *Version VFS, Erweiterung des virtuellen Dateisystems unter Linux um eine Versionsverwaltung*, Universität Potsdam, Dezember 2005.

[20] D. S. H. Rosenthal: *Envolving the Vnode Interface*: USENIX Conference Proceedings, Sommer 1990, S.107-118.

[21] T. W. Page Jr., G. J. Popek, R. G. Guy: *Stackable Layers: An Object-Oriented Approach to Distributed File System Architecture*, University of California Los Angeles, 1991.

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich alle verwendeten Quellen, auch Internetquellen, ordnungsgemäß angegeben habe.

_____

Ort, Datum, Unterschrift