Seminar Langzeitarchivierung Revisionssichere Archivierung - Verteiltes Dokumentenmanagement

John Bechara
1070056
10.05.2009

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

Institut für Softwaretechnologie
Fakultät für Informatik
Universität der Bundeswehr München

Inhaltsverzeichnis

1 Einleitung

Die Leistungsfähigkeit von Computern verdoppelt sich jährlich. Ein Satz, mit dem Gordon Moore, Mitbegründer des bekannten Halbleiterherstellers Intel, berühmt wurde[25]. Und auch wenn "Moore's Law" heute mittlerweile "nur noch" eine 18-24 monatige Verdopplung prognostiziert, so hat doch "dank der Gültigkeit von Moores Gesetz heute jeder PC-Benutzer mehr Rechenleistung auf seinem Schreibtisch stehen als Wissenschaftlern noch 1976, mit dem ersten offiziellen Supercomputer Cray-1, zur Verfügung stand."[12] "Aber das Mooresche Gesetz steht auch für sinkende Kosten."[11] 2007 bspw. betrugen die Kosten für Computer nur noch 2/3 im Vergleich zum Vorjahr[8]. Bereits 2003 waren die Anschaffung und Nutzung von Computern so günstig, dass 80% aller Unternehmen diese einsetzten[7]. Durch diese Entwicklung wurden elektronische Dokumente, und damit auch ihre Ablagesysteme zunehmend wichtiger. Dies ist in besonderem Maße der Fall für die Dokumente, die den bei der Softwareentwicklung anfallenden Source Code beinhalten. So benötigte Microsoft 1993 für Windows NT etwa 4-5 Mio. Source Lines of Codes (SLoC). 10 Jahre später waren für Windows Server 2003 schon 50 Mio. SLoC notwendig. Diese rasante Entwicklung stellt vor allem Programmierer vor große Probleme. Mit zunehmender Größe von Programmen steigt ihre Fehleranfälligkeit und die Anzahl der zur Entwicklung benötigten Programmierer. Erst der Einsatz von sog. Versionskontrollsystemen ermöglicht effektive Gruppenarbeit mit gemeinsam zu bearbeitenden Daten. Das ist auch der Grund dafür, dass Versionskontrollsysteme sich in dieser Domäne am schnellsten verbreitet haben, auch wenn Versionskontrolle theoretisch für eine Vielzahl von Dokumenten möglich ist, wie ich in Kapitel 5 ausführen werde. Folglich befasst sich auch der Großteil der Literatur mit den "Source Code Dokumenten". Aufgrund dieser beiden Faktoren werde ich mich in dieser Seminararbeit schwerpunktmäßig mit den bei der Softwareentwicklung eingesetzten Systemen befassen. Dabei werde ich sowohl auf technische, als auch auf organisatorische Aspekte der Softwareenntwicklung eingehen.
Dazu werde ich in Kapitel 2 auf die allgemeinen Grundlagen von Versionskontrollsystemen eingehen. In den Kapiteln 3 und 4 erläutere ich dann den zentralen Ansatz gefolgt von dem verteilten Ansatz, der in den letzten Jahren stark an Popularität zugenommen hat und Grundlage ist für das verteilte Dokumentenmanagement. Abschließend stelle ich in Kapitel 5 diese beiden Ansätze gegenüber und gehe in Kapitel 6 dann auf die Möglichkeit der Nutzung von verteilten Systemen außerhalb der Softwareentwicklungsdomäne ein.

2 Grundlagen der Versionskontrolle

"Der Begriff Versionskontrolle bezeichnet das Aufzeichnen und Abrufen von Änderungen an einem Projekt"[23]. Ein Versionskontrollsystem (engl. Version Control System bzw. VCS) hat zur Realisierung der Versionskontrolle unter anderem zwei Hauptaufgaben:

- Sicherung und Wiederherstellung:
  Sicherstellen einer Speicherung[19] und der Möglichkeit, "unerwünschte Änderungen zu revidieren"[1].

- Festhalten von Änderungen:
  Welcher Nutzer hat welche Dateien zu welchem Zeitpunkt in welcher Art und Weise geändert[1]?

Zum weiteren Verständniss von Versionskontrollsystemen ist es zunächst notwendig einige Grundbegriffe von Versionskontrollsystemen zu erläutern.

- Repository:
  Das Repository ist der Kern des VCS. Hier werden alle Daten, üblicherweise in Form einer baumartigen Ordner- /Dateistruktur, gespeichert[14].

- Sandbox:
  Die Sandbox ist eine Kopie der Dateien eines Projektes vom Repository. Üblicherweise wird nicht direkt auf dem Repository gearbeitet sondern in der Sandbox, um nicht die Lauffähigkeit der auf dem Repository befindlichen Daten zu gefährden[23]. Dabei kann sich der physikalische Speicherort auf einem Fremdsystem oder auf dem eigenen lokalen System befinden. Im Fall der lokalen Speicherung bezeichnet man die Sandbox auch als "working copy".

- Diff:
  Ein Diff ist ein Prozess, bei dem die Unterschiede zwischen zwei Versionen einer Datei festgestellt werden.[3]

- Delta:
  "Ein Delta ist eine vollständige Beschreibung der Unterschiede zweier Versionen"[9]. Ein Delta ist also das Ergebnis eines Diffs.

- Patch:
  Als Patch wird die Operation des "Anwendens eines Deltas auf ein Dokument" bezeichnet. Das dient der Wiederherstellung einer spezifischen Version eines Dokuments[17].

3 Zentrale Versionskontrollsysteme

Bei zentralen Versionskontrollsystemen (engl.Centralized Version Control Systems bzw. CVCS) handelt es sich um ein Client-Server-Model. Demnach sind die Informationen auf dem Server in Dateien gespeichert und zu jeder Datei existiert eine Liste von Deltas. Man unterscheidet dabei Rückwärts- und Vorwärtsdeltas. Bei Rückwärtsdeltas wird die aktuellste Version als Vollversion gespeichert und die Deltas dienen dazu, vorhergehende Versionen erstellen zu können. In Abbildung 1 sind die Rückwärtsdeltas blau dargestellt. Rückwärtsdelta bieten hier den Vorteil eines direkten Zugriffs auf die aktuelle Version. Allerdings nimmt man u.U. die Speicherung von mehreren Vollversionen in Kauf. Nicht so bei Vorwärtsdeltas, in Abbildung 1 schwarz dargestellt, die nur eine Vollversion speichern, aber deshalb auch für den Aufbau der aktuellsten Version alle Deltas laden müssen[9].

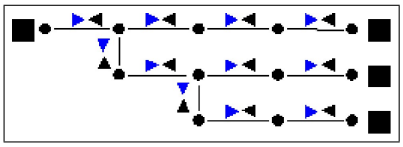

Abbildung 1: Rückwärts- und Vorwärtsdeltas [9]

Durch die Client-Server-Architektur wird auch der Zugriff von mehreren Nutzern auf eine Datei ermöglicht. Das erlaubt ein "quasi-gleichzeitiges" Arbeiten auf der gleichen Datei[14]. "Quasi-gleichzeitig" deshalb, weil zwar alle Nutzer von der gleichen Datei lesen können, aber in der Regel nur auf einer Kopie dieser Datei arbeiten. Befindet sich die Orginal-Datei im Repository, so arbeitet der Nutzer in seiner eigenen Sandbox, die sich üblicherweise auch auf dem Client-Rechner befindet und gleichzusetzen ist mit den in Abbildung 2 dargestellten Working Directories[18].
Berücksichtigt man die Tatsache, dass es, wie in Abbildung 2, viele Working Directories gibt, aber alle nur über ein Repository kommunizieren und ihre Daten austauschen, wird schnell klar, dass diese Vorgehensweise zu Problemen führen kann, die in folgendem Zitat recht deutlich werden: "how will the system allow users to share information but prevent them from accidently stepping on each other´s feet?"[14]

3.1 Das Lock Modify Unlock Model

Um sich eben nicht "gegenseitig auf die Füße zu treten" gibt es generell zwei Lösungsansätze. Der erste ist bekannt als das sog. Lock Modify Unlock Model. Wie der Name schon sagt müssen einzelne Dateien während einer Änderung gesperrt sein, um sie vor fremdem Zugriff zu bewahren.[1] Erst nach Beendigung der Arbeit an den Dateien werden diese wieder freigegeben[16]. Diese Vorgehensweise ist auch als pessimistische Versionskontrolle bekannt[6]. Allerdings gibt es dabei drei wesentliche organisatorische Nachteile. Zum Einen entstehen administrative Probleme. Sollte nämlich ein Nutzer eine Datei sperren und diese Sperrung nicht wieder aufheben, ist es in der Regel mit großem Aufwand verbunden diese Sperrung durch einen Dritten, üblicherweise einen Administrator, wieder aufzuheben. Das kann zum Beispiel passieren wenn das Client-System abstürzt oder vom Nutzer einfach nur vergessen, wurde die Datei wieder freizugeben[14].
Desweiteren ist auch bei Sperrung einer einzelnen Datei oder sogar eines Ordners nicht sichergestellt, dass Konflikte verhindert werden. Das liegt an den üblicherweise vorliegenden Verflechtungen zwischen Dateien. Führen also zwei Nutzer an zwei Dateien ohne Absprache Änderungen durch, können u.U. Konflikte entstehen, die von den Nutzern dann manuell gelöst werden müssten[14]. "Das Sperrsystem konnte dieses Problem nicht verhindern und hat zudem noch eine falsche Sicherheit vorgetäuscht, da beide Benutzer davon ausgingen, dass ihre Änderungen keinen Schaden angerichtet haben."[16]

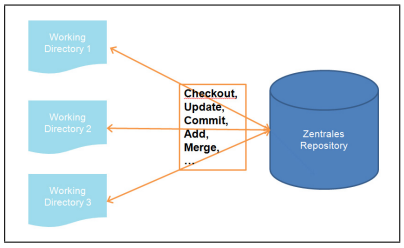

Abbildung 2: Repository

Dazu kommt noch, dass ein serieller Entwicklungsprozess erzwungen wird, obwohl der nicht immer notwendig ist. So können zwei Nutzer beispielsweise unter Absprache an zwei verschiedenen Stellen einer Datei arbeiten, ohne sich notwendigerweise gegenseitig zu behindern. In diesem Fall wäre das Lock Modify Unlock Model äußerst unproduktiv, weil es paralleles Arbeiten nicht zulassen würde, obwohl es meist unproblematisch wäre[14].

[1] Theoretisch sind Locks auch auf anderen Granularitätsstufen, beispielsweise das Sperren eines Abschnittes, möglich. Im Bereich der Softwareentwicklung werden Dateien aber als atomar angesehen. Daher beziehen sich die Locks mindestens auf Dateien. Ein Sperren ganzer Verzeichnisse ist aber auch in diesem Umfeld durchaus möglich und denkbar.

3.2 Das Copy Modify Merge Model

U.a. aufgrund dieser Nachteile wurde das Copy Modify Merge Model entwickelt, welches auch als optimistische Versionskontrolle bekannt ist[16]. Wie beim Lock Modify Unlock Model arbeitet jeder Nutzer in seiner Sandbox und lädt das Ergebnis dann auf den Server hoch. Da bei diesem Modell versucht wird Sperren zu vermeiden, ist es aber möglich, dass ein anderer Nutzer in der Zwischenzeit schon eine neue Version in das Repository eingeführt hat. Dadurch ist es notwendig die Version mit der neueren auf dem Server zusammenzuführen[16]. Ein Beispiel dafür findet sich in Abbildung 3. Sowohl Sue, als auch Joe kopieren die Einkaufsliste r3 vom Repository in ihre Sandbox und arbeiten seperat an ihr. Während Joe aber die Eier durch Käse ersetzt und damit die Version r3*(Joe) erstellt, ersetzt Sue die Eier durch HotDog und erstellt damit die Version r3*(Sue). Da Joe beim Hochladen auf das Repository noch die Version r3 vorfindet, ergeben sich bei ihm keine Probleme und es entsteht die Version r4. Sue allerdings ging bei ihren Änderungen von der Version r3 aus, welche nicht mehr aktuell ist. Es gibt keine Eier mehr, die sie ersetzen könnte. Dadurch ist ein Konflikt entstanden, der beim Zusammenführen der Versionen r4 und r3*(Sue) geloest werden muss. Dieses Zusammenführen bezeichnet man als Verschmelzen (engl. Mergen) und es gibt verschiedene Ansätze für die Umsetzung.
Neben zahlreichen anderen Möglichkeiten lassen sich diese Ansätze unterscheiden in Two-Way- und Three-Way-Merging. Bei Letzterem würden beim Mergen die Informationen des sog. "Nearest Common Ancestor" mitverwendet. Der Nearest Common Ancestor zweier Versionen ist die Version, die sowohl Vorgänger-Version von der einen, als auch von der anderen ist und die gleichzeitig von allen diesen Vorgänger-Versionen am Nächsten zu beiden liegt. In Abbildung 3 wäre der Nearest Common Ancestor von Version r3*(Joe) und r3*(Sue) die Version r3. Der Two-Way-Merge ignoriert diese Information völlig[13].
In Abbildung 3 würde also ein Two-Way-Merge nur die Unterschiede zwischen den Versionen r3*(Joe) und r3*(Sue) vergleichen, und dabei feststellen, dass bei r3*(Joe) in Line 2 Käse steht und in r3*(Sue) an der selben Stelle Hot Dog, während ein Three-Way-Merge erkennen würde, dass in beiden Fällen die Eier durch das jeweilige Wort ersetzt wurden, was einen Informationsvorsprung bedeutet, der vielleicht zum automatischen Lösen dieses Konfliktes führen könnte. Das macht den Three-Way-Merge leistungsfähiger als die Two-Way Variante, in dem Sinne, dass mehr Konflikte gelöst werden können[13].

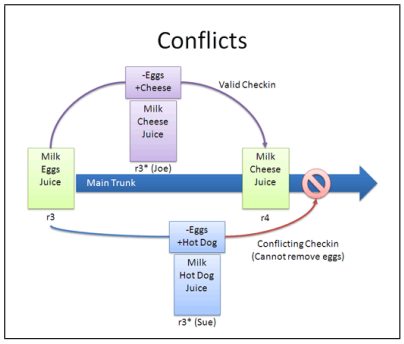

Abbildung 3: Conflicts [3]

Zusätzlich zu diesem rein technischen Ansatz der Konfliktlösung wird auf organisatorischer Ebene versucht Konflikte präventiv zu vermeiden. So ist es bei großen Nutzergruppen üblich, nur wenigen Nutzern Schreibrechte zu geben. Dadurch wird zwar eine zu große Fluktuation auf dem Server vermieden, aber auch die Produktivität stark eingeschränkt. Alle Nutzer, die etwas im Repository Bestehendes ändern wollen, müssen sich an die mit Schreibrechten ausgestatteten Nutzer wenden. Da diese aber auch die möglichen auftretenden Konflikte lösen müssen, was üblicherweise sehr viel Zeit in Anspruch nimmt, kommt es zu zusätzlichen Verzögerungen[21].

3.3 Parallele Entwicklung eines Projekts

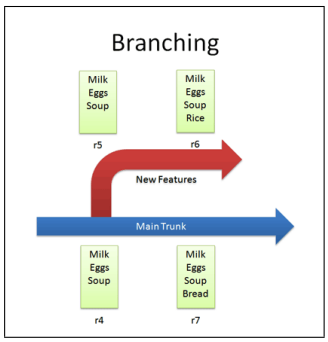

Abbildung 4: Branching [3]

Heutige CVCS, wie zum Beispiel Subversion, bieten die Möglichkeit von der strikten Entwicklungslinie abzuweichen und Verzweigungen einzurichten, die später wieder in den Hauptordner übertragen werden (siehe Abbildung 4)[1]. Diese Zweige nennt man Branches, während der Hauptordner üblicherweise als Trunk bezeichnet wird[16]. So kann man den Trunk zum Release Management verwenden und in ihm immer eine lauffähige Version halten, während man in den Branches arbeitet. Sowohl der Trunk als auch alle Branches befinden sich aber auf dem Server und sind somit für alle zugänglich. Oranisatorisch hat das zur Folge, dass zum Einen paralleles Arbeiten erleichtert wird[1], was die Effizienz fördert, und zum Anderen ein experimentelles Vorgehen bei der Entwicklung ermöglicht wird[23].
Technisch gesehen führen Branches aber in den meisten Fällen zu einem Merge mit dem Trunk, d.h. auch gleichzeitig zu möglichen Konflikten, die es zu lösen gilt. Da Branches oftmals ganze Entwicklungszweige beinhalten, ist es wahrscheinlich, dass der Zeitraum der Erstellung und des Mergens recht groß ist. Auch ist es wahrscheinlich, dass je größer der Zeitraum ist, desto mehr Versionen schon mit dem Trunk gemerged wurden und desto "weiter weg" ist der Nearest Common Ancestor, denn in der Zwischenzeit können viele andere Entwickler Zugriff auf den Trunk genommen und die Version geändert haben. In diesem Fall würde der Mergeprozess recht aufwendig werden und es würden viele manuelle Eingriffe erforderlich sein[15]. Hier kann das Versionskontrollsystem jedoch zumindest Hilfestellung leisten, indem es zum Beispiel die Konflikte markiert, bzw. auch Lösungsvorschläge liefert[1]. Die weitgehend unbefriedigenden Lösungen der eben genannten Probleme beim parallelen Arbeiten und bei den beiden Entwicklungsmodellen waren der Grund für die Entwicklung von Verteilten Versionskontrollsystemen.

4 Verteilte Versionskontrollsysteme

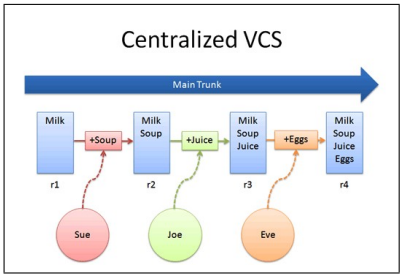

Abbildung 5: CVCS [2]

Verteilte Verrsionskontrollsysteme (engl. Distributed Version Control Systems bzw. DVCS) besitzen prinzipiell dieselben Fähigkeiten wie ihre zentralen Pendants und viel von den oben genannten Techniken findet sich auch hier wieder. Der große Unterschied ist allerdings, dass sich auf dem Nutzer-System nicht nur eine Working Copy befindet, sondern ein eigenes individuelles, voll funktionierendes Repository inklusive History. So erhält jeder Nutzer seinen eigenen Branch. Die Entwicklungslinie, wie sie bei den CVCS, bedingt durch die Client-Server-Architektur, zu finden ist (Abbildung 5), wird zu einem gerichteten azyklischen Graphen (engl. Directed Acyclic Graph, Kurz DAG)[24].

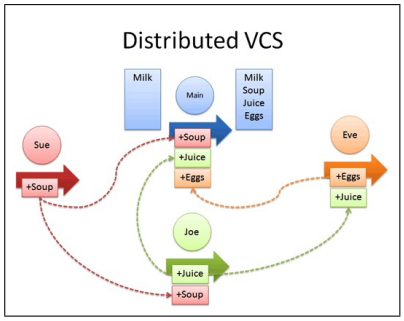

Abbildung 6: DVCS [2]

Abbbildung 6 zeigt ein Beispiel eines Entwicklungsprozesses bei einem DVCS. Sue fügt Suppe zur Einkaufsliste in ihrem Repository hinzu. Das schickt sie sowohl an den Main-Server, wo bisher nur Milch stand, als auch direkt an Joe, der ihre Änderung nun bekommt und seinerseits Saft hinzufügt und nur diese Änderung "Saft " direkt an Eve schickt und auf den Server hochlädt. Die wiederum fügt nun Eier hinzu und lädt diese Änderung auf den Server hoch. Das Resultat ist, dass auf dem Server jetzt die Vereinigungsmenge aller Änderungen gemerged vorliegt, während die Nutzer-Repositories nicht alle Änderungen enthalten. Bei Bedarf kann jetzt die Version vom Server mit der eigenen gemerged werden. Ebenfalls gut zu erkennen ist hier der beim Entwicklungsprozess bei DVCS entstehende DAG.

5 DVCS vs. CVCS

Der verteilte Ansatz bei Versionskontrollsystemen ist durchaus nicht unumstritten. Der Wegfall einer stringenten Entwicklungslinie und der bei diesem Ansatz entstehende DAG wird von den Kritikern als größter Nachteil der DVCS genannt. So gibt es nicht automatisch eine Hauptversion, die immer lauffähig ist. In einem Unternehmen würde man durch die Nutzung von DVCS viel Kontrolle über den Source Code verlieren. Die meisten anderen DVCS bieten aber die Möglichkeit einen solchen Main-Server einzurichten[10], denn ein zentraler Speicherort für das Release Management ist für die meisten Softwareprojekte zwingend erforderlich[20]. So kann bei einem Open Source Projekt die Community den Master festlegen und von allen Teilnehmern verlangen mit diesem zu arbeiten[5]. Zusätzlich hat man bei einem solchen Verfahren auch den Vorteil, dass nicht auf dem Server gearbeitet wird. Er dient also rein dem Verteilen von Releases. Die eigentliche Arbeit wie z.B. das Merging oder die Konfliktbehebung wird unabhängig vom Main-Server gemacht.
Linus Torvalds hat in seinem Vortrag über Git bei Google sogar die Idee einer Qualitätsmanagement Gruppe eingebracht. Diese Qualitätsmangement Gruppe wäre dann zum Einen der zentrale Service Desk für alle Daten, die in das Master-Repository eingefügt werden sollen. Zum Anderen hätte sie aber auch die Aufgabe über die verschiedenen Zweige zu iterieren, mögliche Releases für das Master-Repository zu identifizieren, ähnliche Zweige zu verbinden, aber auch Abbruch-Entscheidungen für nicht mehr relevante Zweige zu treffen. Es ist hier sogar denkbar, dass verschiedene Zweige im Master-Repository abgelegt werden. Denn selbst wenn zu irgendeinem Zeitpunkt aus bestimmten Gründen eine Abbruch-Entscheidung getroffen wurde, ist es möglich, dass diese Gründe zu einem späteren Zeitpunkt wegfallen und der Zweig vielleicht wieder aufgenommen werden kann. So könnte das Master-Repository neben dem Release Management auch zum Ideen Management verwendet werden[21].
Demgegenüber stehen sowohl technische, als auch organisatorische Vorteile. Aus technischer Sicht befindet sich das Repository auf der Festplatte. Es ist also, wie oben bereits genannt, möglich, fast alle Operationen offline durchzuführen. Das schließt auch das Mergen und die Konfliktbehebung mit ein. Auch für das Austauschen von Deltas ist keine lange Internetverbindung erforderlich[4]. Viele DVCS, wie zum Beispiel Git, bieten die Möglichkeit diese einfach per Email zu verschicken. Das sorgt für eine deutlich bessere Performance im Vergleich zu den CVCS, denn diese müssen für jede Operation auf dem Repository eine Kommunikationsbeziehung über ein Netzwerk zum Server herstellen. Es sind vor allem diese Netzwerkoperationen, die bei einem CVCS viel Zeit benötigen und bei einem DVCS wegfallen[21].
Das bringt gleichzeitig auch den Vorteil der Unabhängigkeit von einer zen-

tralen Instanz, denn "verwaltet [..] ein zentraler Server in einer verteilen Umgebung die Zugriffe auf eine gemeinsame Ressource, so können Probleme entweder im Ausfall des Servers - da die Ressource im Netz nicht mehr verfügbar ist - auftauchen, oder aber der Server kann zum Engpaß bei Zugriffen werden."[22]
Organisatorisch hat jeder Entwickler seine eigene Sandbox. Fehlgeschlagene Programmierversuche werden, anders als bei Branches, nicht öffentlich. Dadurch wird das Experimentieren mit neuen Ideen und kreatives Arbeiten gefördert[2]. Es entfallen auch alle weiteren Probleme, die bei CVCS mit Branches entstehen würden.
Die Entwickler können ohne Berücksichtigung hierarchischer Strukturen miteinander kommunizieren. Die Kommunikation findet, wie bereits oben erwähnt, direkt zwischen den Nutzern statt, ohne die Notwendigkeit Branches zu beantragen oder sich an Nutzer mit Schreibrechten zu wenden. Das fördert die Teamarbeit und beschleunigt dadurch die Entwicklung[4].
Des Weiteren wird aus dem Pull-Prinzip der CVCS (jeder Nutzer muss schauen ob Änderungen auf dem Server vorliegen) ein Push-Prinzip bei den DVCS, jeder Nutzer bekommt die Änderungen zugeschickt, bspw. per Mail, und kann selbst entscheiden ob und wann er diese Änderungen einbringt.
Auch wird einem oben genannten großen Nachteil von CVCS, der Überlastung der wenigen Nutzer mit Schreibrechten auf dem zentralen Server, durch DVCS entgegengewirkt. Linus Torvalds hat es in seinem Vortrag so ausgedrückt: "That´s what it´s all about. They did all the work for me."[21]
Der Entwickler übernimmt das Mergen und die Konfliktbehebung. Das ist auch insofern sinnvoll, dass ein Entwickler das, was er produziert hat, auch am besten Mergen und die Konflikte am besten beheben kann; ein deutlich effizienteres Arbeiten ist möglich.

6 DVCS für Dokumente

Auch wenn ich mich bei dieser Arbeit weitestgehend auf die Software Entwicklung und die damit verbundenen Source Code Management Systeme (SCMS) beschränkt habe, spricht nichts gegen eine darüber hinausgehende Nutzung bei anderen Dokumenten in anderen Domänen. Diese Dokumente müssten allerdings einige Anforderungen erfüllen. So arbeiten die beschriebenen SCMS zeilenbasiert. Sie vergleichen also bei einem Diff Zeile für Zeile und speichern die Unterschiede dann in einem Delta. Andere Dokumente, wie zum Beispiel Office Dokumente, werden dagegen als Binäre Datein gespeichert. Zwar gibt es auch bei diesen die Möglichkeit eines Diffs, allerdings mündet dieser Diff nicht in einem Delta, sondern dient nur dazu dem Nutzer die Unterschiede zwischen zwei Versionen zu zeigen. Die Versionen werden dabei vollständig gespeichert[17]. Ein Dokument, welches sich für verteilte

Systeme eignet, müsste also mindestens die oben bereits genannten Operationen "diff", "delta" und "patch" unterstützten.
Wie bei den Source Code Dokumenten würden auch alle anderen Dokumente von den Vorteilen der DVCS profitieren. Auch hier würde paralleles Arbeiten erleichtert werden. Man wäre auch unabhängig von einem zentralen System, was zu einer Ausfallsicherheit führen würde. Und Änderungen würden durch das Push-Prinzip früh erkannt und durch die einfache Verbreitung per Email leicht zu verteilen sein.

7 Fazit

Die steigenden Anforderungen, die damit zunehmende Komplexität und die dadurch bedingte wachsende Anzahl der beteiligten Entwickler machen Alternativen zu den bisherigen CVCS zwingend notwendig. Verteilte Systeme haben zahlreiche Vorteile gegenüber ihren zentralen Pendants. Sie ermöglichen offline Arbeiten, bieten erhöhte Performance und fördern die Teamarbeit und kreatives Arbeiten.
Zusätzlich dazu ist auch denkbar, dass der Main-Server eines DVCS-basierten Projektes, aufgrund der strikten Beschränkung auf das Release Management auch zur Archivierung der Dokumente verwendet wird, wobei eine Beschränkung des Dokumenttypes nur durch die in Kapitel 6 angesprochenen Anforderungen besteht.
DVCS stellen allerdings noch eine neue Technik dar und bisher wurden wenige Erfahrungen damit gemacht. Für die Verwendung von DVCS bei Dokumenten außerhalb der Softwareentwicklungs-Domäne gibt es auch noch kaum Forschungsergebnisse oder Literatur. Des Weiteren sind die Anforderungen, die an eben diese Dokumente gestellt werden außerhalb der Softwareentwicklung kaum umgesetzt. Speziell in diesem Bereich ist also noch Forschung nötig und es bleibt abzuwarten, ob und wann die verteilten Systeme ihre zentralen Pendants vollständig ablösen werden.

Literatur

[1] Stefan Baerisch. Versionskontrollsysteme in der softwareentwicklung. Technical report, Informationszentrum Sozialwissenschaften der Arbeitsgemeinschaft Sozialwissenschaftlicher Institute e.V. (ASI), 2005.

[2] BetterExplained. Intro to distributed version control (illustrated). Website. Available online at `http://betterexplained.com/articles/intro-to-distributed-version-control-illustrated/`; visited on Mai 8th 2009.

[3] BetterExplained. A visual guide to version control. Website. Available online at `http://betterexplained.com/articles/a-visual-guide-to-version-control/`; visited on Mai 8th 2009.

[4] Ian Clatworthy. Distributed version control systems why and how. Website. Available online at `http://people.ubuntu.com/~ianc/papers/dvcs-why-and-how.xhtml`; visited on Mai 8th 2009.

[5] Ben Collins-Sussman. The risks of distributed version control. Website. Available online at `http://blog.red-bean.com/sussman/?p=20`; visited on Mai 8th 2009.

[6] Dominik Denker. Versionsverwaltung mit cvs bzw. subversion. Website. Available online at `http://syspect.informatik.uni-oldenburg.de/archiv/doc/CVSandSVN.pdf`; visited on Mai 8th 2009.

[7] Statistisches Bundesamt Deutschland. 80% der unternehmen setzten im jahr 2003 computer ein. Website. Available online at `http://www.destatis.de/jetspeed/portal/cms/Sites/destatis/Internet/DE/Presse/pm/2004/03/PD04__138__ikt.psml`; visited on Mai 8th 2009.

[8] Statistisches Bundesamt Deutschland. Cebit 2007: Prices of it goods falling, with performance increasing. Website. Available online at `http://www.destatis.de/jetspeed/portal/cms/Sites/destatis/Internet/EN/press/pr/2007/03/PE07__106__614.psml`; visited on Mai 8th 2009.

[9] Jürgen Ebert. Universität koblenz, skript zur vorlesung softwaretechnik ii. Website. Available online at `http://www.uni-koblenz.de/~espi/Softwaretechnik2.pdf`; visited on Mai 8th 2009.

[10] Robert Fendt. Dvcs round-up: One system to rule them all?–part 1. Website. Available online at `http://ldn.linuxfoundation.org/article/dvcs-round-one-system-rule-them-all-part-1`; visited on Mai 8th 2009.

[11] Intel. Vierzig jahre mooresches gesetz. Website. Available online at http://www.intel.com/cd/corporate/techtrends/emea/deu/209836.htm; visited on Mai 1st 2009.

[12] Sascha Kurz. *Konstruktion und Eigenschaften ganzzahliger Punktmengen*. PhD thesis, Universität Bayreuth, 2005. Available online at http://www.mathe2.uni-bayreuth.de/sascha/papers/diss.pdf; visited on Mai 1st 2009.

[13] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[14] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion*. O'Reilly, 2. edition, 2008.

[15] Martin Steigerwald Reinhard Tartler. Markt modell - verteilte versionsverwaltung mit bazaar. *Linux-Magazin*, (06), 2007.

[16] Markus Röhling. Praxisprojekt-konfigurationsmanagement-template am beispiel von antmod. Website. Available online at http://et.fh-duesseldorf.de/c_personen/a_professoren/lux/prax_bachelor/PraxisprojektRoehling.pdf; visited on Mai 8th 2009.

[17] Sebastian Rönnau, Jan Scheffczyk, and Uwe Borghoff. Towards xml version control of office documents. In *DocEng '05: Proceedings of the 2005 ACM symposium on Document engineering, ACM, New York, NY, USA*, pages 10–19, 2005.

[18] Jan Schütze. Verteilte versionsverwaltung - ein vergleich von darcs, git, mercurial und bazaar. Website. Available online at http://www.fh-wedel.de/~si/seminare/ws08/Ausarbeitung/06.vvw/vvw1.htm; visited on Mai 8th 2009.

[19] Harvey P. Siy Thomas Ball, Jung-Min Kim Adam A. Porter. If your version control system could talk ... Website. Available online at http://research.microsoft.com/en-us/um/people/tball/papers/icse97-decay.pdf; visited on Mai 8th 2009.

[20] Linus Torvalds. Re: clarification on git, central repositories and commit access lists. Website. Available online at http://lwn.net/Articles/246381/; visited on Mai 8th 2009.

[21] Linus Torvalds. Versionsverwaltung mit subversion, tech talk: Linus torvalds on git. Speech. Available online at http://www.youtube.com/watch?v=4XpnKHJAok8; visited on Mai 8th 2009.

[22] Johann Schlichter Uwe Borghoff. *Rechnergestützte Gruppenarbeit: Eine Einführung in verteilte Anwendungen.* Springer, 1998.

[23] Jennifer Vesperman. *CVS.* O'Reilly Germany, 2004.

[24] Tommi Virtanen. Tv's cobweb: Git for computer scientists. Website. Available online at `http://eagain.net/articles/git-for-computer-scientists/`; visited on Mai 8th 2009.

[25] Wikipedia. Mooresches gesetz. Website. Available online at `http://de.wikipedia.org/wiki/Mooresches_Gesetz`; visited on Mai 1st 2009.