Monotone - Ein Brevier für verteilte Versionsverwaltung

Autor: Mathias Weidner
Datum: 2011-07-01
Basisrevision: e8e5949711762d9f644baaa7bb4371b7162a8bcf
Aktuelle Revision: e8e5949711762d9f644baaa7bb4371b7162a8bcf

Inhaltsverzeichnis

Vorwort

Im Jahre 2007 arbeitete ich nach langer Zeit wieder an einer Hochschule in einem Forschungsprojekt. Was mir auffiel, war, dass viele Mitarbeiter und auch Studenten der Sektion Informatik kaum Kenntnisse über die Verwendung von Versionsverwaltungssystemen (RCS) hatten. Da ich seit etlichen Jahren mit CVS und davor mit anderen Systemen gearbeitet hatte, begann ich als erstes nach einem geeigneten Versionsverwaltungssystem für das Projekt zu suchen. CVS und auch andere Systeme mit einem zentralen Repository fielen aus, da wir für das Projekt keinen durchgängig laufenden Server zur Verfügung hatten. Ich hatte ohnehin schon seit einiger Zeit von verteilten Systemen für die Versionsverwaltung (DVCS) gehört. Das sollte der Einstieg werden. Ich bin tief verwurzelt in der UNIX-Welt und damit stand mir eigentlich eine große Palette zur Verfügung. Die anderen Mitarbeiter am Projekt arbeiteten jedoch mit MS-Windows, also war ein natives Windows-Programm Pflicht, möglichst mit Installer. Da wir in Java entwickelten, kam uns entgegen, dass es auch ein Java-Projekt mit monotone gab, durch das wir monotone mit Eclipse und Apache Ant verwenden konnten. So kam ich zu monotone, das ich seither schätzen gelernt habe. Im folgenden Jahr begann ich, eine kleine Anleitung für monotone zu schreiben. Diese blieb sehr lange liegen, bis ich 2011 genügend Zeit und Schwung hatte, es zu diesem Text auszubauen.

Für wen ist dieses Buch

Wie schon erwähnt, soll das eine Einführung in die Arbeit mit monotone sein. Die Zielgruppe sind Leute, die überhaupt noch nicht mit oder bisher mit anderen Versionsverwaltungssystemen gearbeitet haben. Wer bereits längere Zeit mit monotone gearbeitet hat, wird vielleicht noch die eine oder andere Anregung im Kapitel *Anpassungen und Erweiterungen* finden.

Wie ist das Buch aufgebaut

Im Kapitel *Grundlagen* geht es ganz kurz darum, wofür ich monotone verwende, was es für mich macht und wo meine Daten hinkommen.

Beim nächsten Kapitel *Installation* geht es nur darum, wie man monotone auf seinen Rechner bekommt.

Das Kapitel *In Gang kommen* beschreibt die ersten Schritte mit monotone und wie man es lokal einsetzt.

Im nächsten Kapitel *Verteilt über das Netz* geht es um das, was verteilte Versionsverwaltungssysteme aus macht, die Arbeit mit mehreren Repositories und die Synchronisation zwischen diesen.

Das Kapitel *Tag für Tag* behandelt verschiedene Problemstellungen, die bei der täglichen Arbeit mit monotone vorkommen können.

Das nächste Kapitel *Datensicherung* hätte auch gut in das vorige hineingepasst. Ich habe es nur wegen der Wichtigkeit des Themas - ich arbeite seit mehr als zehn Jahren als Systemadministrator - als eigenes Kapitel herausgehoben.

Beim Kapitel *Anpassungen und Erweiterungen* geht es zum einen um die Script-Sprache Lua und wie man damit monotone Dinge beibringen kann, die es so noch nicht kann. Zum anderen um externe Programme, die - automatisch aufgerufen - mit monotone zusammenarbeiten.

Damit sind wir schon beim Anhang.

Im Anhang 0 versuche ich einige der verwendeten Begriffe, so wie ich sie verstanden habe, zu erklären.

Anhang 1 listet weiterführende Informationen auf. Das sind überwiegend URLs von Webseiten, über die man einzelne Themen vertiefen kann.

In Anhang 2 stelle ich einige Programme vor, die mit monotone zusammenarbeiten.

Anhang 3 und 4 sind Kurzfassungen aus Themen in diesem Text.

Zur Schreibweise

Thomas Keller schrieb mir: *monotone sollte im Satz immer kleingeschrieben werden, lediglich am Satzanfang groß.* Daran halte ich mich hier. Das Programm selbst heißt `mtn`.

Für Programm-Beispiele und Eingaben auf der Kommandozeile verwende ich eine dicktengleiche Schrift. Diesen nehme ich auch im Fließtext, wenn ich Optionen wortgetreu, das heißt so, wie es eingegeben werden könnte, verwende.

Ansonsten verwende ich einen *kursiven* Font für Hervorhebungen von Namen, auch von Parametern. So kann es vorkommen, das ich an einer Stelle vom monotone-Befehl *sync* schreibe und an anderer `mtn sync`. Beide Male ist das gleiche gemeint.

Danksagung

Dieses Buch wäre nicht entstanden, wenn ich nicht so viele Leute getroffen hätten, die nicht nur keine Versionsverwaltung verwenden, sondern sich aktiv dagegen sträuben, *weil der Aufwand das zu Lernen nicht durch den Nutzen*

aufgewogen wird. Da es mir schwerfällt, diese Leute vom Nutzen der verschiedenen Versionsverwaltungssysteme zu überzeugen - ich bin nicht so gut als Verkäufer - hoffe ich wenigstens, den Aufwand etwas verringert zu haben.

Besonders dankbar bin ich für die Hinweise, die ich über die Mailingliste *monotone-users* bekommen habe. Insbesondere von Marc Lütolf und Thomas Keller. Weiterhin erhielt ich Korrekturen und Anregungen von Stefan Naumann.

Grundlagen

> Wenn es nicht in der Versionsverwaltung ist, existiert es nicht.
>
> —Troy Hunt

Monotone als Versionsverwaltungssystem hält für mich Zwischenstände von Dateien, die ich bearbeite, bereit. Da sind zum einen die aktuellen Versionen im Arbeitsbereich und die jeweils mit dem Vorgänger und Nachfolgern verketteten Versionen im Repository.

Das Repository ist dabei die Stelle, an der Buch geführt wird über alle von mir bestätigten Änderungen an den Dateien im Arbeitsbereich. Bei monotone ist das Repository eine SQLite-Datenbank. Da monotone ein verteiltes Versionsverwaltungssystem ist, gibt es nicht nur ein Repository sondern mehrere, üblicherweise eins auf dem lokalen Rechner welches im Arbeitsbereich referenziert wird und weitere Repositories auf anderen Rechnern, die mit dem lokalen abgeglichen werden.

Eine Datei ist im Repository jeweils mit ihrem Vorgänger und Nachfolger (soweit vorhanden) verknüpft. Dabei ist es möglich, dass eine bestimmte Version keinen Vorgänger hat (es ist die erste eingecheckte Version dieser Datei), einen Vorgänger (Datei wurde ausgecheckt, geändert, eingecheckt) oder mehrere (die Version ist aus der Vereinigung verschiedener älterer Versionen entstanden). Ebenso ist es möglich, dass eine Version keinen Nachfolger hat (es ist die letzte eingecheckte Version), einen Nachfolger, oder mehrere (die Version wurde mehrfach ausgecheckt, verschieden editiert und wieder eingecheckt).

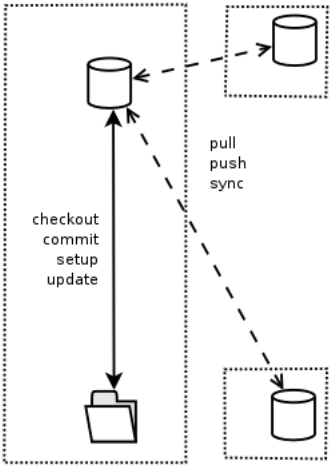
Um einen bestimmten Stand verschiedener Dateien, sozusagen einen Schnappschuss, im Repository festzuhalten, verwendet monotone interne textbasierte Datenstrukturen, die ursprünglich Manifest genannt wurden. Diese selbst werden ebenfalls im Repository verwaltet. Das Manifest, welches seit Version 0.26 Roster genannt wird, enthält den relativen Pfad, die zugehörige Version und noch weitere Informationen zu jeder erfassten Datei zu diesem Zeitpunkt.

Alle im Repository verwalteten Daten werden mit kryptografischen Schlüsseln signiert und dadurch gegen Veränderung gesichert.

Wo sind meine Daten

Die Daten, die von monotone verwaltet werden, befinden sich an vier verschiedenen Orten:

- im *Keystore* befinden sich meine privaten Schlüssel
- im *Arbeitsverzeichnis* sind die Dateien, die ich unmittelbar bearbeite
- im *lokalen Repository* befinden sich die gespeicherten Revisionen der Dateien
- in einem *entfernten Repository* sind die Revisionen der von mir und eventuell anderen bearbeiteten Dateien, nachdem mit dem Repository synchronisiert wurde.



Die folgenden Befehle bewegen meine Daten in und zwischen den verschieden Orten.

Mit den monotone-Kommandos *setup* und *checkout* kann ich einen neuen lokalen Arbeitsbereich einrichten. Mit *add*, *rm* und *rename* kann ich Dateien zum überwachten Arbeitsbereich hinzufügen, daraus entfernen oder umbenennen.

Der Befehl *update* bringt Änderungen aus dem lokalen Repository in meinen Arbeitsbereich. Mit *commit* übernehme ich Änderungen im Arbeitsverzeichnis in das lokale Repository. Mit *revert* kann ich Änderungen im Arbeitsverzeichnis rückgängig machen. Diese Befehle beeinflussen lediglich das lokale Repository und meinen Arbeitsbereich.

Erst mit dem monotone-Kommando *pull* hole ich Daten aus einem entfernten Repository in das lokale und mit *push* schiebe ich Daten aus dem lokalen Repository in das entfernte, *sync* schließlich gleicht die Unterschiede zwischen lokalem und entfernten Repository in einem Aufruf aus, das heißt, lokale Änderungen werden in das entfernte Repository übernommen und umgekehrt.

Für *pull* wird kein Schlüssel benötigt, wenn der Server entsprechend konfiguriert ist. Damit ist ein anonymer Download möglich.

Schließlich gibt es noch das monotone-Kommando *genkey*, das einen neuen Schlüssel erzeugt und diesen im Keystore ablegt.

Installation

Bevor es losgehen kann, müssen wir uns das Programm beschaffen, was nicht sehr schwierig ist. Die neueste Version gibt es bei http://www.monotone.ca/. Die Beispiele sind alle mit Version 1.0 getestet.

Debian GNU/Linux und Ubuntu Linux

Prinzipiell kann man monotone bei Debian GNU/Linux mit dem Befehl `apt-get` installieren. In der momentan stabilen Version von Debian (6.0 Squeeze) wird dann aber Version 0.48 von monotone installiert, in der aktuellen LTS-Version von Ubuntu (10.04 Lucid Lynx) gar 0.45, sodass ich hier die Installation aus den Quellprogrammen beschreibe.

Zuerst müssen wir ein paar notwendige Programme und Bibliotheken installieren, die für die Übersetzung von monotone benötigt werden:

```
$ sudo apt-get install autoconf automake bzip2 gettext g++   \
                       libboost-dev libz-dev libbotan1.8-dev \
                       libsqlite3-dev libpcre3-dev           \
                       liblua5.1-0-dev libidn11-dev          \
                       libgmp3-dev libbz2-dev stow texinfo
```

Als Nächstes holen wir uns die Quellprogramme und entpacken diese:

```
$ wget http://www.monotone.ca/downloads/1.0/monotone-1.0.tar.bz2
$ tar xjf monotone-1.0.tar.bz2
```

Wir begeben uns in das entpackte Verzeichnis mit den Quellprogrammen und konfigurieren monotone für unseren Rechner mit `configure`. Wenn wir nichts angeben, wird monotone nach */usr/local/bin* installiert. Ich installiere selbst kompilierte Software aber lieber nicht direkt dorthin, sondern gehe einen kleinen Umweg mit dem Programm `stow`, der es mir erleichtert selbst kompilierte Software selektiv zu entfernen, oder zwischen verschiedenen Versionen umzuschalten:

```
$ cd monotone-1.0
$ ./configure --prefix=/usr/local/stow/monotone-1.0
```

```
$ make
$ make check
$ sudo make install
$ sudo stow -d /usr/local/stow -v monotone-1.0
```

Nun ist monotone unter */usr/local/stow/monotone-1.0* installiert und in */usr/local/bin*, */usr/local/etc* und */usr/local/share* gibt es symbolische Links dorthin. Wir können monotone direkt aufrufen:

```
$ mtn --version
```

Microsoft Windows

Für Microsoft Windows gibt es einen Installer, der die Installation vereinfacht.

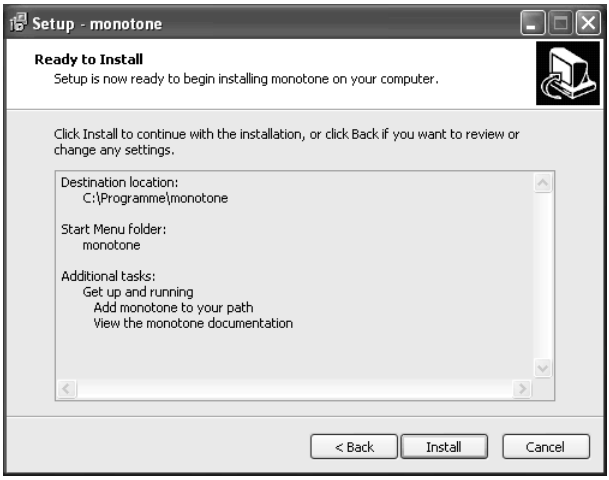
Zunächst akzeptieren wir die Lizenz (GPL, GNU General Public License) und dann geht es los.

Wir wählen das Installationsverzeichnis und können hier noch nachschauen, ob dort ausreichend Platz verfügbar ist.

Dann können wir den Folder im Startmenü benennen, bzw. entscheiden, ob wir überhaupt einen wollen.

Danach markieren wir, dass monotone zum Suchpfad für ausführbare Programme hinzugefügt wird und entscheiden, ob wir die Dokumentation gleich oder später (bzw. gar nicht) sehen wollen.

Schließlich werden uns alle ausgewählten Optionen noch einmal angezeigt und dann geht es los.



Das folgende Dialogfenster zeigt uns, dass monotone auf dem Rechner installiert ist.

In Gang kommen

Um mit monotone zu arbeiten, benötige ich mindestens ein lokales Repository, einen Schlüssel um die abgespeicherten Daten zu signieren und einen Arbeitsbereich, dessen Inhalt von monotone verfolgt wird.

Ein lokales Repository anlegen

Um Dateien mit monotone zu verwalten, muss ich ein lokales Repository anlegen. Prinzipiell kann man ein Repository für mehrere oder alle Projekte verwenden, ich ziehe in den meisten Fällen ein Repository für ein Projekt vor. Das lokale Repository lege ich mit folgendem Befehl an:

```
$ mtn --db /pfad/zum/projekt.mtn db init
```

Alternativ kann ich bei neueren Versionen von monotone ein lokales Repository automatisch beim Klonen eines entfernten Repositorys mit dem Befehl *clone* anlegen:

```
$ mtn --db /pfad/zum/project.mtn clone server branch
```

Hierbei ist *project.mtn* der Name der neuen Datenbankdatei, *server* die Adresse des entfernten Repository und *branch* der Name des Zweiges, den ich laden will.

Ich habe bei beiden Varianten den Pfad zur Projektdatenbank expliziert angegeben, weil es mir hier darum ging, Möglichkeiten, ein lokales Repository[1] anzulegen, aufzuzeigen. Bei neueren monotone-Versionen ist es möglich, auf die Angabe der lokalen Datenbank zu verzichten. In diesem Fall verwendet monotone die Standarddatenbank *:default.mtn*. Neuere Versionen verwenden das Verzeichnis *$HOME/.monotone/databases* beziehungsweise unter MS Windows *%APPDATA%\monotone\databases* für Datenbanken, deren Name mit einem Doppelpunkt (**:**) beginnt. Wird keine Datenbank angegeben, so wird die Standarddatenbank genommen.

[1] Die Datenbank könnte irgendwo abgelegt werden, bei mir landeten früher meist alle im Verzeichnis *~/A/monotone*. Seitdem ich die Notation *:datenbankname* kenne, lege ich immer mehr Datenbanken im Verzeichnis *$HOME/.monotone/databases* ab.

Persönliche Schlüssel erzeugen

Mit meinen persönlichen Schlüsseln signiere ich die Daten,[2] die ich im Repository einspeichere. Über diesen Schlüssel werden auch die Zugriffsrechte auf entfernte Repositories eingestellt:

```
$ mtn genkey mathias@example.net
```

Um das Kennwort für den Schlüssel nicht jedes Mal eingeben zu müssen, kann ich (unter UNIX, mit installiertem ssh_agent) den Schlüssel zur Verwaltung an ssh_agent übergeben:

```
$ mtn ssh_agent_export ~/.ssh/id_monotone
$ chmod 600 ~/.ssh/id_monotone
$ ssh-agent /bin/bash
$ ssh-add ~/.ssh/id_monotone
```

Anschließend brauche ich den monotone-Schlüssel nur noch einmal während der Sitzung eingeben.

Unter MS Windows wird der ssh-agent von *putty* unterstützt.

Alternativ (und weniger sicher) kann ich das Kennwort in einen Hook in der Datei *monotonerc*[3] eintragen:

```
$ mkdir ~/.monotone
$ cat >> ~/.monotone/.monotonerc
function get_passphrase(keypair_id)
    return "geheim"
end
^D
```

Den öffentlichen Schlüssel benötige ich, um Änderungen von anderen Entwicklern zu überprüfen und um Zugriffsrechte für Repositories festzulegen. Um an den öffentlichen Schlüssel zu kommen, gebe ich das Folgende ein:

```
$ mtn pubkey mathias@example.net
```

Mit diesem Befehl gibt monotone den öffentlichen Schlüssel so in Textform aus, dass er in andere Datenbanken mit `mtn read` eingelesen werden kann.

---

[2] Vom Nutzer signiert werden die Zertifikate (author, branch, changelog, date, suspend, tag, testresult, ...). Alle anderen Daten sind unsigniert.

[3] Die Datei *monotonerc* bzw. Ihr Äquivalent unter MS Windows ist üblicherweise im Verzeichnis *$HOME/.monotone* bzw. unter MS Windows in *%APPDATA%\monotone* zu finden. Hier gibt es auch ein Unterverzeichnis *keys*, in dem die monotone bekannten Schlüssel gespeichert sind, und das Unterverzeichnis *databases*, in dem die von monotone verwalteten Datenbanken liegen.

Einen Arbeitsbereich anlegen

Einen Arbeitsbereich (workspace) kann ich anlegen indem ich einen vorhandenen Zweig aus einem lokalen Repository auschecke (`mtn checkout`) oder indem ich einen neuen Arbeitsbereich aufsetze (`mtn setup`). So lege ich einen neuen Arbeitsbereich an:

```
$ mkdir -p ~/P/projekt
$ cd ~/P/projekt
$ mtn --db /pfad/zum/projekt.mtn \
      setup --branch projekt.branch
```

Monotone kann nun im lokalen Repository */pfad/zum/projekt.mtn* Dateien aus dem Verzeichnis *~/P/projekt/* im Zweig *projekt.branch* verwalten. Von sich aus tut es das allerdings nicht. Damit monotone den aktuellen Stand einer Datei im lokalen Repository verwaltet, muss ich ihm die Datei vorher bekannt machen.

Dateien hinzufügen / umbenennen / entfernen

Dateien im Arbeitsbereich, die von monotone verwaltet werden sollen, kann ich ihm mit `mtn add` bekannt machen. Mit `mtn drop` schließe ich eine Datei von der weiteren Verwaltung durch monotone aus (ältere Versionen bleiben im Repository erhalten und zugänglich). Wenn ich eine Datei umbenennen oder in ein anderes Verzeichnis verschieben und dabei ihre Versionsgeschichte erhalten möchte, verwende ich `mtn rename`.

Welche Dateien monotone verwaltet (kennt), zeigt mir das Kommando `mtn list known` im Arbeitsbereich. Dem entsprechend zeigt `mtn list unknown` die Dateien im Arbeitsbereich, die monotone nicht kennt (nicht verwaltet). Schließlich zeigt `mtn list ignored` die Dateien, die monotone gar nicht kennen will. Diese letzte Liste kann ich mit einer Datei namens *.mtn-ignore* im Arbeitsbereich beeinflussen[4]. Damit kann ich die Dateien, deren Änderungen ich nicht verfolgen will, ausschließen und schließlich mit `mtn add --unknown` alle anderen Dateien mit einem Mal unter Versionsverwaltung stellen.

Änderungen bestätigen

Monotone merkt sich nahezu beliebig viele Zwischenstände der von ihm verwalteten Dateien, aber nur auf Aufforderung mit `mtn commit`. Dieser Befehl verlangt eine Notiz, die die damit erzeugte Revision beschreibt. Die Notiz kann

[4] Natürlich kann ich auch *.mtn-ignore* mit monotone verwalten.

ich mit der Option `--message` auf der Kommandozeile mitgeben. Ansonsten öffnet monotone einen Texteditor, mit dem man die Notiz in aller Sorgfalt erstellen kann. Erst dann wird der momentane Stand aller verwalteten Dateien im lokalen Repository zwischengespeichert.

Zweige

Zu Beginn der Arbeit an einem Projekt bilden die verschiedenen Versionen meiner Dateien einen geraden, nicht verzweigten, gerichteten Graphen, der jeweils vom Vorgänger zum Nachfolger in einer Kette geht. Man spricht hier auch von Trunk-Version.

Wenn ich bereits einen oder mehrere Stände des Projekts veröffentlicht habe, wird es früher oder später passieren, das ich einen Fehler in einer schon veröffentlichten Version bereinigen muss, die weit zurückliegt. Das ist ein guter Moment um einen neuen Zweig im Repository zu starten, der genau bei der veröffentlichten Version vom Stamm abzweigt. Monotone pflegt für mich diesen alternativen Entwicklungszweig im selben Repository und bezieht unabhängig von den Änderungen in der Trunk-Version alle Änderungen in diesem Zweig auf die Version, mit der er gestartet wurde.

Ein anderer Grund, einen neuen Zweig einzurichten, wäre zum Beispiel um ein neues Feature auszuprobieren, für das bei der Entwicklung in der Trunk-Version keine Zeit ist.

Der einfachste Weg, einen neuen Zweig anzulegen, ist den Namen des neuen Zweiges beim Commit anzugeben:

```
$ mtn --branch trunk.neuerzweig commit
```

Alternativ kann man den neuen Zweig schon vor dem nächsten Commit anlegen:

```
$ mtn cert h: branch trunk.neuerzweig
$ mtn update --branch trunk.neuerzweig
```

Obwohl das etwas aufwendiger ist, hat es doch den Vorteil, das man später beim Commit nicht mehr daran denken muss, den Zweig anzugeben, da die Dateien in meinem Arbeitsverzeichnis nun bereits zum neuen Zweig gehören und `mtn commit` automatisch in diesem aktualisiert.

Änderungen zusammenführen

Wenn in einem Zweig verschiedene Änderungen von einem oder mehreren Entwicklern bestätigt wurden, kann es passieren, dass es mehrere letzte Stände

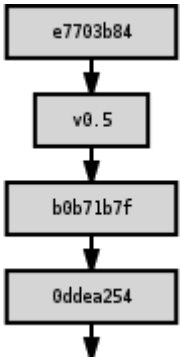
gibt. Diese führe ich mit `mtn merge` zusammen. Nach der nächsten Synchronisation der Repositories haben alle wieder den gleichen Stand.

Möchte ich hingegen die Änderungen eines Zweiges auf einen anderen Zweig anwenden, verwende ich `mtn propagate`.

Versionen kennzeichnen

Ein Problem bei monotone ist für uns Menschen eine bestimmte Revision zu identifizieren, da bei monotone jede Revision nach dem kryptografischen Schlüssel (SHA1) der Datei benannt ist, die den Stand ausmacht.

Wenn ich weiß, dass ich eine bestimmte Revision später wieder erkennen will, zum Beispiel weil ich diese Revision veröffentliche, dann markiere ich diese Revision mit einem *version tag*.



Um einen Stand zu markieren, bestätige ich die letzten Änderungen mit `mtn commit`, informiere mich über die aktuelle Version mit `mtn head` und markiere diesen Stand mit `mtn tag <revision> <tag>`.

Ein Repository von CVS übernehmen

Mitunter habe ich ein älteres Projekt, das ich mit CVS - einem älteren, in vielen Jahren bewährten, mittlerweile aber an einige Grenzen stoßenden Versionsverwaltungssystem - gepflegt hatte. Nun wäre es einfach, die letzten stabilen Stände und die Entwicklungsstände zu nehmen und ein neues Repository mit monotone anzufangen.

Hier gibt es einen besseren Weg. Wenn ich Zugriff auf das ganze Repository (auf die Dateien, nicht nur ein Netzwerkzugriff über den CVS-Server) habe, kann ich die komplette Historie des Projektes übernehmen. Das Feature wird

zwar auf der Homepage von monotone als “noch etwas unreif” bezeichnet, bei meinen Projekten funktionierte es immer recht gut. Das Importieren kann allerdings - insbesondere bei größeren Projekten - etwas länger dauern. Das ist dem Umstand geschuldet, dass CVS auf Basis von einzelnen Dateien arbeitet, das heißt, alle Versionsinformationen sind immer nur an die betreffende Datei gebunden, während monotone mit Verzeichnisbäumen arbeitet. Um das Manifest für eine Revision zu erstellen, vergleicht monotone daher das Commit-Datum und die Commit-Notizen aller Dateien. Das ist nicht trivial und braucht etliche Rechenzeit. Laut Homepage sind aber schon CVS-Repository in der Größe von etwa einem GB erfolgreich importiert worden - meine Projekte waren alle kleiner.

Die Vorgehensweise ist wie folgt:

```
$ mtn --db=projekt.mtn db init
$ mtn --db=projekt.mtn genkey cvsimport@example.net
$ mtn --db=projekt.mtn --branch net.example.abc \
        cvs_import /usr/local/cvsroot/abc
$ mtn --db=projekt.mtn --branch net.example.abc \
        checkout abc-neu
```

Als Erstes erzeuge ich eine neue Datenbank und einen Schlüssel für den CVS-Import. Für den Schlüssel kann ich auch einen vorhandenen nehmen, besser aber nicht den, mit dem ich normalerweise Zertifikate signiere, damit die Herkunft leichter unterschieden werden kann. Der monotone-Befehl `cvs_import` liest das Repository ein. Dabei ist */usr/local/cvsroot* das Verzeichnis, welches bei CVS in der Umgebungsvariable *$CVSROOT* verwendet wird und *abc* der Name des CVS-Projektes, welcher gleichzeitig der Name des Unterverzeichnisses unter *$CVSROOT* ist. Nach dem Import von CVS hole ich mir den letzten Stand in den Arbeitsbereich *abc-neu*.

Verteilt über das Netz

Im vorigen Kapitel haben wir uns mit der grundlegenden Arbeitsweise von monotone vertraut gemacht. Wir sind nun in der Lage, private Projekte unter Revisionsverwaltung zu stellen. Der nächste Schritt ist die Arbeit mit mehreren Repositories, die über das Netz synchronisiert werden.

Vorbereiten

Alle im Repository abgelegten Daten werden mit kryptografischen Schlüsseln signiert. Das ist ein grundlegendes Element dieses Versionsverwaltungssystems. Wenn ich gemeinsam mit anderen arbeiten will, muss ich deren Schlüssel kennen und akzeptieren, genauso, wie sie meinen Schlüssel kennen und akzeptieren müssen. Der erste Schritt zur Zusammenarbeit ist daher der Austausch der Schlüssel:

```
$ mtn pubkey mathias@example.net > mathias.pubkey
```

Für meinen eigenen Schlüssel muss ich dabei keine Datenbank angeben. Ich kann diesen Schlüssel nun an alle Mitarbeiter schicken, die mir ihre öffentliche Schlüssel senden und meinen Schlüssel importieren:

```
$ mtn --db lokales/repository.mtn read < mathias.pubkey
```

Beim Import muss immer eine Datenbank angegeben werden, weil das Vertrauen gegenüber fremden Schlüsseln immer projektbezogen ist. Nun kennt monotone also die Schlüssel aller Beteiligten, das reicht aber noch nicht ganz.

Damit der monotone-Server Daten, die mit anderen (wenn auch bekannten) Schlüsseln signiert wurden, akzeptiert, muss ich diesen Schlüsseln noch ein paar Rechte einräumen:

```
$ cat >> ~/.monotone/read-permissions <<EOT
pattern "*"
allow "mathias@example.net"
EOT
```

Dadurch wird mir mit dem Schlüssel *mathias@example.net* erlaubt, alle Zweige in dem betreffenden Repository zu lesen, das heißt in ein anderes Repository zu kopieren (zum Beispiel mit `mtn pull`).

Außerdem brauche ich noch Schreibzugriff, damit ich meine Daten in das Repository schicken kann (`mtn push` bzw. `mtn sync`):

```
$ cat >> ~/.monotone/write-permissions <<EOT
mathias@example.net
EOT
```

Bei den Schreibrechten wird nicht nach Zweig unterschieden.

Damit haben wir die Vorbereitungen abgeschlossen und können einen monotone Server starten, mit dem die anderen Mitarbeiter ihre Repositories synchronisieren können.

Abgleich über Netz

Dieser Teil ist nun wieder sehr simpel. Damit andere sich mit meinem monotone-Repository abgleichen können, starte ich den Server wie folgt:

```
$ mtn --db lokales/repository.mtn serve
```

Damit lauscht monotone am TCP-Port 4691 und bietet alles aus der Datenbank `lokales/repository.mtn` entsprechend den in `read-permissions` und `write-permissions` eingestellten Regeln an.

Um sich Daten von diesem Server zu holen, gibt man ein:

```
$ mtn --db anderes/repository.mtn pull rechner.von.mathias "*"
```

Dabei kann man statt `"*"` auch direkt den Namen des Zweigs angeben oder ein geeignetes Muster, das alle gewünschten Zweige erfasst.

Wenn man schon einen eigenen, eventuell geänderten Stand hat, will man eher beide Repositories auf den gleichen Stand bringen:

```
$ mtn --db anderes/repository.mtn sync rechner.von.mathias "*"
```

beziehungsweise:

```
$ mtn --db anderes/repository.mtn sync
```

Der zweite Aufruf ist möglich, wenn bereits einmal mit dem Rechner synchronisiert wurde und dieser als `default-server` in der Datenbank vermerkt ist (siehe `mtn list vars`).

Wenn ich nur Änderungen auf den Server schieben will, nehme ich:

```
$ mtn --db anderes/repository.mtn push
```

Die Anmerkungen wie zu `mtn sync` gelten auch hier.

Zentraler Server

Ein Grundprinzip von verteilten Versionsverwaltungssystemen ist, dass ein zentraler Server nicht mehr nötig ist.

Wenn mehrere Leute an einem Projekt arbeiten, insbesondere wenn die Mitarbeiter räumlich oder zeitlich getrennt sind, ist es trotzdem praktisch ein Repository zu haben, das immer an der gleichen Stelle über das Netz zu erreichen ist. Dann wird dieses wie ein zentrales Repository verwendet und der Abgleich zwischen den Entwicklern kann darüber erfolgen. Ist der zentrale Server einmal nicht zu erreichen, ist immer noch der direkte Abgleich zwischen den Entwicklern möglich.

Ein weiterer Vorteil eines zentralen Servers ist der, dass man zum Synchronisieren nur noch `mtn sync` eingeben muss und den Namen von Server und Zweig weglassen kann, wenn die betreffenden Variablen in der Datenbank passend eingestellt sind. Konkret sind das die Variablen `database: default-server` und `database: default-include-pattern`. Diese sehen in etwa so aus:

```
$ mtn list vars
database: default-exclude-pattern
database: default-include-pattern net.example.abc
database: default-server monotone.example.net
known-servers: monotone.example.net b1c3782090a03536fc21f9f...
```

Die Variable `known-servers: monotone.example.net` enthält den Schlüssel für den monotone-Server. Dieser Schlüssel wird nicht für das Signieren der Dateien im Repository sondern nur für die Kommunikation mit dem Server verwendet. Falls sich der Schlüssel auf dem Server ändert, teilt mir das monotone bei der nächsten Verbindungsaufnahme mit. Damit ist die Gefahr geringer, dass ich mit dem falschen Server synchronisiere.

Unter Debian GNU/Linux gibt es ein Paket `monotone-server`, welches Scripts enthält, die das Aufsetzen eines monotone-Servers vereinfachen.

Der Server benötigt keine eigene Datensicherung, da alle Versionen, die er enthält, auch auf den Rechnern der Entwickler vorhanden sind. Bei einem Ausfall kann ein neues Repository jederzeit aus einem der anderen Repositories, welches alle Zweige und die letzten Änderungen enthält, neu aufgesetzt werden. Lediglich der private Schlüssel für den Server sollte an einem sicheren Ort verwahrt werden, damit beim neuen Aufsetzen mit dem gleichen Schlüssel weitergearbeitet werden kann und die beteiligten Entwickler keine Meldung über geänderte Serverschlüssel bei der Synchronisation bekommen.

Anonymer Download

Insbesondere bei Open Source Entwicklungen möchte man ein Repository gern zum einfachen Download der Quellen freigeben. Das wird in `read-permissions` eingestellt:

```
$ cat >> ~/.monotone/read-permissions <<EOT
pattern "net.example"
allow "*"
EOT
```

Damit können alle Zweige, die mit *net.example* beginnen, mit `mtn pull` gelesen werden. Egal mit welchem Schlüssel.

Andere Transportwege

Die Synchronisation der Datenbanken von monotone basiert auf einem Protokoll namens *netsync*. Normalerweise transportiert monotone dieses Protokoll über eine einfache TCP-Verbindung. Das ist nicht der einzige Transportweg, über den monotone *netsync* verwenden kann. Es ist möglich dieses Protokoll über *SSH* oder irgendein Programm, das eine Full-Duplex-Verbindung aufbauen kann, zu verwenden.

Wenn monotone mit dem *pull*, *push* oder *sync* Befehl aufgerufen wird, reicht es das erste Argument an einen Lua-Hook weiter, der es in einen Befehl zur Verbindungsaufnahme umwandeln soll. Wenn dieser Hook einen Befehl zurückliefert, startet monotone diesen lokal und spricht das *netsync* Protokoll über die Standardeingabe und -ausgabe des entsprechenden Processes.

Wenn der Lua-Hook keinen Befehl liefert, versucht monotone das erste Argument als TCP-Adresse zu interpretieren (ein Name mit einer optionalen Portnummer), baut eine TCP-Verbindung dorthin auf und spricht *netsync* über diese Verbindung.

Monotone versteht per Default drei URI-Schemata[5]:

- SSH-URIs der Form `ssh://[user@]hostname[:port]/pfad/zur/db.mtn`

  Hierbei wird monotone auf dem fremden Rechner via SSH aufgerufen und das *netsync* Protokoll über die Standardein- und -ausgabe verwendet.

  Bei einer SSH-URI muss das Programm *ssh* lokal im Suchpfad für ausführbare Programme zu finden sein und das Program *mtn* auf dem entfernten Rechner.

[5] siehe Anhang A0: URI

- Datei-URIs in der Form `file:/pfad/zur/db.mtn`

  Hierbei wird monotone auf dem lokalen Rechner aufgerufen mit der angegebenen Datenbank als Argument zu `--db`. Das *netsync* Protokoll wird wie bei SSH-Verbindungen über die Standardein- und -ausgabe verwendet.

  Bei einer Datei-URI muss das Programm *mtn* lokal im Suchpfad für Programme zu finden sein.

- monotone(-TCP)-URIS in der Form `mtn://hostname[:port]`

  Hierbei baut monotone eine TCP-Verbindung zu dem angegebenen Rechner und Port 4691 bzw. dem angegebenen Port auf und verwendet das *netsync* Protokoll darüber.

Sowohl bei SSH-URIs als auch bei Datei-URIs wird die Datenbank gesperrt, das heißt man benötigt Schreibrechte auf die Datenbank, auch wenn man nur mit *pull* Daten holt und die Datenbank nicht dauerhaft modifiziert. Außerdem wird für diese beiden Transporte die Default-Authentisierung deaktiviert.

Um weitere Transportwege zu unterstützen, muss man die beiden Lua-Hooks `get_netsync_connect_command` und `use_transport_auth` anpassen.

Tag für Tag

Arbeitsablauf

Bei der täglichen Arbeit mit monotone (oder auch anderen VCS) gibt es etliche mögliche Arbeitsabläufe um die Änderungen im Code zu verwalten.

Einer davon ist, im Trunk-Zweig nur umfangreiche Änderungen wie zum Beispiel neue Features oder Bugfixes zu führen und für die detaillierten Änderungen (die eigentliche Arbeit an den Features oder Bugfixes) jeweils eigene Zweige zu nehmen. Verwendet man eine Fehler-Datenbank oder ein Ticketsystem, so kann man den Namen für die Arbeitszweige daraus ableiten (zum Beispiel die Fehler- oder Ticketnummer).

Dieser Arbeitsablauf ist dem in http://nakedstartup.com/2010/04/simple-daily-git-workflow/ beschriebenen Arbeitsablauf für *git* (ein anderes verteiltes Versionskontrollsystem) nachempfunden.

Daraus ergibt sich folgender grundlegender Ablauf für jedes Ticket, jeden Fehler:

1. Checkout mit neuem Zweig
2. Arbeit am Problem mit vielen Commits
3. Merge des neuen Zweigs mit Trunk

Diese drei Abläufe sehen im Einzelnen wie folgt aus.

Checkout mit neuem Zweig

Entweder ich halte alle von mir bearbeiteten Tickets in demselben lokalen Repository:

```
$ mtn --db alles.mtn pull
$ mtn --db alles.mtn co -branch trunk
```

Oder ich nehme jedesmal ein frisches und checke den aktuellen Stand aus:

```
$ mtn --db ticketnr.mtn db init
$ mtn --db ticketnr.mtn pull remotehost --branch trunk
$ mtn --db ticketnr.mtn co --branch trunk .
```

Als Nächstes stelle ich auf den neuen Zweig um:

```
$ mtn cert h: branch trunk.ticketnr
$ mtn update --branch trunk.ticketnr
```

Damit bin ich bereit für die

Arbeit am Problem mit vielen Commits

1. Arbeit am Problem
2. `mtn list unknown`
3. `mtn add | move | remove`
4. `mtn status | diff`
5. `mtn commit -m "detaillierte Beschreibung"`
6. wieder von vorn, bis das Problem gelöst ist

Wenn alle Tests in dem neuen Zweig fehlerfrei durchlaufen, bin ich so weit, die Änderungen wieder in den Trunk-Zweig einzupflegen.

Merge des neuen Zweigs mit Trunk

Je nachdem, wie viel Leute am Projekt mitarbeiten, ist die Wahrscheinlichkeit dafür, dass inzwischen andere Änderungen in trunk eingepflegt wurden hoch oder niedrig. Darum hole ich als Erstes den aktuellen Stand von trunk in mein lokales Repository und spiele etwaige Änderungen von trunk in meinen Zweig ein:

```
$ mtn pull
$ mtn propagate trunk trunk.ticketnr
```

Damit habe ich die Chance mögliche Konflikte in meinem Zweig aufzulösen, ohne den Trunk-Zweig mit Zwischenständen zu belasten.

Ich lasse noch mal alle Tests durchlaufen. Bei Fehlern kehre ich zurück zur Arbeit am Problem in meinem Zweig. Ging alles gut, schiebe ich meine Änderungen in den Trunk-Zweig und veröffentliche sie[6]:

```
$ mtn propagate trunk.ticketnr trunk
$ mtn sync
```

Wenn zwischen dem `mtn pull` und dem `mtn sync` niemand anders Änderungen in den Trunk-Zweig eingespielt hat, bin ich fertig.

In dem wahrscheinlich eher seltenen Fall, dass jemand in der Zwischenzeit wieder eine neue Version in Trunk eingespielt hat, haben wir nun vielleicht

[6] Mit `mtn sync` veröffentliche ich meine Änderungen im entfernten Repository und hole mir gleichzeitig die Änderungen, die dort von anderen veröffentlicht wurden.

zwei oder mehr Endversionen im Trunk-Zweig. Diese vereinige ich in meinem lokalen Repository mit:

```
$ mtn merge trunk
```

Dabei löse ich eventuell auftretende Konflikte und lasse noch mal alle Tests durchlaufen. Bei Problemen propagiere ich den letzten Stand des Trunk-Zweiges noch mal in meinen Problemzweig und kehre zurück zur Arbeit am Problem.

Ging alles gut, veröffentliche ich die Änderungen:

```
$ mtn sync
```

Ein anderes Modell wäre zum Beispiel, dass nur einer der Entwickler Änderungen mit dem Trunk-Zweig abgleicht und dazu vorher seinen Zweig mit den Änderungen der anderen Entwickler aktualisiert.

Wenn ein Stand erreicht ist, der eingefroren und veröffentlicht werden soll, markiere ich diese Revision mit einem *version tag* und erzeuge einen neuen Zweig, in dem nur noch Fehlerkorrekturen für diesen veröffentlichten Stand gesammelt werden.

Konflikte beim Zusammenführen

Es gibt zwei Möglichkeiten, verschiedene Endrevisionen eines Zweiges zusammenzuführen. Vor dem Commit der eigenen Änderungen mit `mtn update` und nach dem Commit mit `mtn merge`. Bei beiden Varianten verwendet monotone die gleichen Algorithmen zum Zusammenführen. Bei der ersten Variante ist es möglich, dass bei einem Konflikt, der nicht gleich manuell gelöst werden kann, der Arbeitsbereich einen fehlerhaften Stand hat. Die zweite Variante hat demgegenüber den Vorteil, dass man bei Problemen mit dem Zusammenführen auf die bestätigte, nicht zusammengeführte Version zurückgehen und weiterarbeiten kann. Wenn der Konflikt gelöst ist, kann man die Revisionen immer noch zusammenführen.

Konflikte können beim Zusammenführen zweier Revisionen in der Datenbank oder beim Zusammenführen im Arbeitsbereich auftreten.

Der monotone-Befehl *show_conflicts* zeigt Konflikte, die auftreten, wenn man das Revisionen in der Datenbank (nach *commit*) zusammenführen will. Leider kann dieser Befehl noch nicht die Konflikte im Arbeitsbereich auflisten.

Mit dem Befehl *conflicts* und seinen Unterbefehlen können Konfliktlösungen für monotone vorgegeben werden. Dieser Befehl erfordert einen Arbeitsbereich. Die Konfliktlösungen werden in einer Datei gespeichert und bei `mtn merge` verwendet.

Konflikt-Typen

- Konflikte im Inhalt von Dateien

  Diese kommen am häufigsten vor. Sie treten zum Beispiel auf, wenn in einer Datei unterschiedliche Änderungen gegenüber dem gemeinsamen Vorfahren aufgetreten sind. Bei Änderungen an verschiedenen Zeilen wird monotone diese Änderungen intern zusammenführen.

  Falls binäre Dateien betroffen sind, oder die gleiche Zeile auf verschiedene Weise geändert wurde, ruft monotone den *merge3* Hook auf. In der Default-Implementierung schreibt dieser die verschiedenen Texte in temporäre Dateien und ruft das externe Programm auf, das von der Lua-Funktion *get_preferred_merge3_command()* angegeben wird, um den Konflikt zu lösen.

- Doppelte Namen

  Dieser Konflikt tritt auf, wenn zwei Dateien oder Verzeichnissen in beiden Revisionen der gleiche Name gegeben wurde. Diese Dateien oder Verzeichnisse können neu aufgenommen oder in den widersprechenden Namen umbenannt worden sein.

  Die Lösung für diesen Konflikt hängt von den genaueren Umständen ab.

  – Gleiche Datei

    In diesem Fall wird die Datei in einer der Revisionen mit entfernt. Der monotone-Befehl *conflicts* hilft hierbei.

  – Verschiedene Dateien

    Hier muss die Datei in einer der Revisionen umbenannt werden. Das zieht für gewöhnlich Änderungen in anderen Dateien, die diese Datei referenzieren, nach sich. Auch hier hilft `mtn conflicts ...`

  – Verzeichnisse

    Auch hier gibt es die beiden Basis-Strategien *Entfernen* und *Umbenennen*. Da beim Entfernen eines Verzeichnisses auch alle Dateien mit entfernt werden müssen, ist es fast immer besser, das Verzeichnis temporär umzubenennen und die darin enthaltenen Dateien einzeln zu betrachten, indem man sie zusammenführt, umbenennt oder löscht. Schließlich kann man das temporäre Verzeichnis entfernen. Der monotone-Befehl *conflicts* kann hierbei aber nicht helfen, das muss man direkt tun.

- Missing Root

Das tritt eher selten auf. Es passiert, wenn ein Verzeichnis in einer der beiden Revisionen zum Stammverzeichnis gemacht wurde und in der anderen Revision gelöscht wurde.

Dieser Konflikt wird gelöst, in dem in der Revision mit dem geänderten Stammverzeichnis ein anderes Verzeichnis zur Wurzel gemacht wird. In der Revision, in der das Verzeichnis gelöscht wurde, kann man nichts machen, da eine Datei oder ein Verzeichnis, welches einmal (im Repository) gelöscht wurde, nicht wiederbelebt werden kann, sondern nur neu angelegt.

- Ungültige Namen

  Monotone reserviert den Namen *_MTN* im Stammverzeichnis des Arbeitsbereiches für interne Zwecke und behandelt alle Dateien diesen Namens im Stammverzeichnis als illegal für die Versionsverwaltung.

  In anderen Verzeichnissen im Arbeitsbereich ist der Name legal.

- Endlosschleifen bei Verzeichnissen

  Das tritt auf, wenn in einer Revision ein Verzeichnis unter ein zweites verschoben wurde und in der anderen Revision das zweite unter das erste.

  Das kann einfach gelöst werden, indem eines der Verzeichnisse aus dem anderen heraus verschoben wird.

  Der monotone-Befehl *conflicts* kann hierbei noch nicht helfen.

- Verwaiste Knoten

  Ein verwaister Knoten tritt auf, wenn in einer Revision ein Verzeichnis gelöscht wurde und in der anderen Revision Dateien oder Verzeichnisse dort neu angelegt oder dahin verschoben wurden.

  Das kann einfach gelöst werden, indem man die verwaisten Knoten aus dem gelöschten Verzeichnis woandershin verschiebt oder löscht.

  `mtn conflicts` kann hierbei helfen. Falls es sich jedoch bei dem verwaisten Knoten um ein Verzeichnis handelt und dieses gelöscht werden soll, muss erst der Inhalt des Verzeichnisses gelöscht werden, bevor `mtn conflicts` aufgerufen wird.

- Mehrere Namen

  Dieser Konflikt tritt auf, wenn eine Datei in beiden Revisionen in unterschiedliche Namen umbenannt wurde.

  Zur Lösung benennt man die Datei in einer Revision so um, wie sie in der anderen heißt.

  Der monotone-Befehl *conflicts* kann hierbei noch nicht helfen.

- Konflikte von Attributen

  Das passiert, wenn die Attribute einer Datei bzw. eines Verzeichnisses in den beiden Revisionen unterschiedlich sind.

  Zur Lösung ändert man die Attribute in einer oder beiden Revisionen so, dass sie gleich sind.

  Der monotone-Befehl *conflicts* kann hierbei noch nicht helfen.

Fehlersuche durch Zweiteilung

Es ist gute Praxis, bei der Arbeit möglichst kleine Änderungen als eigene Revisionen abzuspeichern. Ein Vorteil ist, dass der Übergang zwischen zwei Revisionen überschaubar bleibt und damit eine eventuell nötige Fehlersuche erleichtert wird.

Dann kann es aber vorkommen, dass bei einem im Laufe der Entwicklung eingeführten Fehler sehr viele Revisionen zwischen der letzten guten und der Revision, bei der der Fehler bemerkt wurde, liegen. Hier hilft das Bisektionsverfahren die Änderung zu ermitteln, bei der der Fehler wahrscheinlich eingeführt wurde.

Die Bisektion benötigt einen Arbeitsbereich ohne Änderungen. Sie startet indem Revisionen mit `mtn bisect good -r rev_good` und `mtn bisect bad -r rev_bad` markiert werden. Damit hat man die Menge der verdächtigen Revisionen eingegrenzt. Monotone wählt als Nächstes die Revision in der Mitte zwischen der guten und der schlechten Revision aus und lädt diese in den Arbeitsbereich. Nachdem man diese Revision getestet hat, markiert man diese ebenfalls mit *good* bzw. *bad* (dabei braucht keine Revision mehr angegeben werden). Monotone halbiert im Arbeitsbereich automatisch den Bereich zwischen der zuletzt als *good* und der zuletzt als *bad* markierten Revision, sodass man auf schnellst möglichem Weg zu der Revision kommen kann, in der der Fehler zum ersten Mal auftrat. Kommt man auf dem Weg hierhin zu einer Revision, die nicht testbar ist (zum Beispiel weil sie nicht kompiliert werden kann), überspringt man diese Revision mit `mtn bisect skip`.

Mit `mtn bisect status` kann man sich jederzeit den Status der Bisektion ausgeben lassen.

Zugriffsrechte

Bei monotone ist es viel wichtiger, was aus einem Repository herauskommt, als was hinein geht. Der Austausch über Netzwerk ist eine einfache Kommunikation von Fakten (Inhalt von Dateien und Revisionen) und Zusicherungen

über deren Wert (die Zertifikate, mit denen diese signiert sind). Diesen Zusicherungen muss nicht notwendigerweise geglaubt werden, im Abschnitt Vertrauensgrundlagen wird näher darauf eingegangen. Es gibt simple Möglichkeiten der Steuerung für den grundlegenden Zugriff auf die Datenbank, alles darüber hinausgehende sollte über Vertrauensverhältnisse geregelt werden.

Konkret kann man Leserechte für den Zugriff über das Netz in der Datei ~/.monotone/read-permissions vergeben und Schreibrechte dementsprechend in der Datei ~/.monotone/write-permissions. Während die Leserechte auf bestimmte Zweige beschränkt werden können, werden Schreibrechte für die ganze Datenbank vergeben.

Ein Beispiel für die Datei read-permissions könnte so aussehen:

```
pattern "net.example.*"
allow "*"
continue "true"

pattern "*"
allow "mathias@example.net"
```

Dies würde aller Welt Lesezugriff auf alle Zweige erlauben, die mit *net.example.* anfangen und mir Zugriff auf alle Zweige.

Die Datei write-permissions enthält lediglich eine Aufzählung von zugelassenen Schlüsseln, einen pro Zeile:

```
mathias@example.net
```

Damit der entsprechende Schlüssel anerkannt wird, muss er vorher mit `mtn pubkey` extrahiert und dann mit `mtn read` importiert werden.

Vertrauensgrundlagen

In monotone wird alles Vertrauen über Zertifikate gehandhabt.

Zertifikate und Revisionen

Jede im Repository abgelegte Version wird mit einer kryptografischen SHA1-Hash identifiziert, die sowohl den Inhalt als auch ihre ganze Abstammung repräsentiert. Ein Zertifikat ist bei Monotone eine RSA-Public-Key-Signatur über ein Tupel aus Revisions-ID, den Namen und den Wert des Zertifikatstyps.

Die normalen Zertifikate, die mit allen Revisionen einhergehen sind für den Autor, den Zweig der Revision, das Änderungsprotokoll (*changelog*) sowie das Datum, welches automatisch hinzugefügt wird.

Das Zweigzertifikat ist das wichtigste für die automatischen Operationen von monotone.

Vertrauenswürdige Versionen

Jeder Benutzer kann seine eigenen Vorlieben darüber konfigurieren, welchen Zertifikaten er vertraut und welchen nicht. Diese Präferenzen können in verschiedenen Arbeitsbereichen unterschiedlich sein. So ist es möglich, in einem Arbeitsbereich den stabilen Zweig zu verfolgen und in diesem Arbeitsbereich zusätzliche Zertifikate von der Qualitätssicherung zu verlangen, während ich in einem anderen Arbeitsbereich die aktuellsten Entwicklungen im Entwicklerzweig verfolge und Zertifikate von allen bekannten am Projekt beteiligten Entwicklern akzeptiere.

Diese Einstellungen werden über Lua-Hooks realisiert.

Synchronisation, Zusammenführen und Vertrauen

Bei monotone sind die Synchronisation, das Zusammenführen (Merge) und das Aussprechen von Vertrauen voneinander getrennt. Das heißt:

- Bei einer Synchronisation werden lediglich Dateien und Zertifikate ausgetauscht.
- Nach der Synchronisation kann ich entscheiden, was ich mit den neuen Revisionen anfange. Zum Beispiel könnte ich diese mit meiner Arbeit zusammenführen, auschecken oder einfach ignorieren.
- Vertrauen gegenüber den neuen Zertifikaten wird erst geprüft, wenn ich meinen Arbeitsbereich aktualisiere.

Qualitätssicherung

Monotones Mechanismen zur Qualitätssicherung beruhen überwiegend auf der Einschränkung von Subgraphen der Revisionsgeschichte. Zwei Möglichkeiten gibt es, diese einzuschränken:

- Durch Einschränken der Menge der vertrauenswürdigen Zweigzertifikate kann man fordern, dass festgelegte Gutachter jede Kante des Subgraphen evaluiert haben.

  Hierfür verwendet monotone die Lua Hooks `get_revision_cert_trust`. Ein Beispiel für dessen Anpassung ist im Abschnitt *Trust Evaluation Hooks* des Handbuches zu finden.

  Die Gutachter können Ihre Bewertung bestimmter Revisionen mit den monotone-Befehlen *approve*, *disapprove*, *suspend* ausdrücken.

- Indem man die Menge der geforderten *testresult* Zertifikate angibt, kann man verlangen, dass die Endpunkte eines *update* Befehls ein Zertifikat haben, welches angibt, dass die betreffende Revision bestimmte Tests oder eine Testreihe erfüllt haben.

  Hierfür verwendet monotone den Lua Hook `accept_testresult_change`, der durch den monotone-Befehl *update* aufgerufen wird. Die Default-Implementierung liefert `true`, wenn die Datei `_MTN/wanted-testresults` nicht existiert. Andernfalls sollte diese Datei eine Liste von Schlüssel-IDs enthalten, die die Testergebnisse repräsentieren. Dann liefert der Hook `false`, wenn eine gelistete Schlüssel-ID sowohl in der alten als auch in der neuen Revision enthalten ist und in der alten Revision `true` ist und in der neuen `false`, andernfalls liefert die Default-Implementierung `true`.

  Ein Testergebnis kann mit dem monotone-Befehl *testresult* hinzugefügt werden. Sinnvollerweise verwendet man für jeden Test bzw. jede Testreihe, die vermerkt werden soll, einen eigenen Schlüssel.

Datensicherung

Bei der Datensicherung sind zwei Dinge wesentlich. Zum einen die Daten selbst mit ihrer Historie, Verknüpfungen und Kommentaren. Zum anderen die Schlüssel, mit denen die Daten signiert wurden. Dazu kommen noch die Einstellungen im Verzeichnis *~/.monotone/*, wie zum Beispiel read-permissions, write-permissions, Lua-Hooks, ...

Im Allgemeinen ist die Sicherung der Daten (nicht der Schlüssel) eher unkritisch, wenn mehrere Leute zusammenarbeiten und daher auch mehrere verteilte Repositories bestehen. Die letzte Revision der Daten erhält man immer automatisch, wenn man mit einem Repository synchronisiert, welches diese Revision enthält. Das sollte jedoch nicht als Freibrief verstanden werden, auf eine Datensicherung ganz zu verzichten. Wenn jeder sich darauf verlässt, dass sich jemand anders darum kümmert, ist im schlimmsten Fall keine Sicherung verfügbar. Wenn gerade das Repository mit dem letzten Stand verloren geht und dieser Stand noch nicht synchronisiert wurde, ist auch dieser verloren.

Die Sicherung der Schlüssel ist aus dem Grunde wichtig, dass bei der normalen Arbeit zwar mit dem Namen des Schlüssels (üblicherweise eine E-Mail-Adresse) gearbeitet wird, monotone intern aber immer den Schlüssel selbst verwendet. Es kann daher in einer Datenbank immer nur einen Schlüssel mit einem Namen geben. Geht dieser Schlüssel verloren, kann man zwar einen neuen Schlüssel anlegen und dem neuen Schlüssel die entsprechenden Rechte vergeben, man muss aber dafür einen anderen Namen (dann eine andere E-Mail-Adresse) geben.

In neueren monotone-Versionen werden Schlüssel nicht mehr über ihren Namen identifiziert, sondern über ihren Hash. Damit ist es auch möglich, einen Schlüssel mit einem Namen zu erzeugen, der bereits von einem alten, verloren gegangenen Schlüssel genutzt wird. Da die Schlüssel weiterhin meistens über ihren Namen visuell identifiziert werden, gibt es das Konzept von “local names”, die per Lua-Hook bestimmten Schlüssel gegeben werden können. Diese Namen werden dann statt dem ursprünglich vergebenen Namen des Schlüssels angezeigt.

Lokale Datensicherung

Wenn ich nur lokal arbeite oder mich nur um mein lokales Repository kümmere, brauche ich die Datei mit dem privaten Repository, meine privaten Schlüssel

und eventuell weitere Einstellungen aus *$HOME/.monotone/* (dort befinden sich auch die Schlüssel). Wichtig ist, dass zum Zeitpunkt der Sicherung das Repository nicht verwendet wird. Wenn ich meine Datenbank in dem von Monotone standardmäßig verwendeten Verzeichnis abgelegt habe (das heißt, ich habe bei der Angabe der Datenbank immer ein **:** vorangestellt), reicht es das Verzeichnis *$HOME/.monotone* (*%APPDATA%\monotone* bei MS Windows) komplett zu sichern.

Zur Wiederherstellung kopiere ich die Dateien wieder an ihren alten Platz und kann sofort beim Stand der letzten Sicherung weiterarbeiten.

Datensicherung über Netz

Eine elegante Art die Daten zu sichern besteht darin, sie mit einem entfernten Repository zu synchronisieren. Ich überlasse die Verantwortung für die Sicherung dann den Betreibern des entfernten Repository.

Auch dann muss ich meine Schlüssel sichern. Einstellungen aus *~/.monotone/*, so vorhanden auch.

Zur Wiederherstellung kopiere ich die Schlüssel und Einstellungen wieder nach *~/.monotone/*, lege eine leere Datenbank an und hole mir die Daten vom Server:

```
$ cp -a pfad/zum/backup/.monotone ~/.monotone
$ mtn --db lokale_db.mtn db init
$ mtn --db lokale_db.mtn pull remote.server "zweigmuster"
```

Automatische Sicherung eines monotone-Servers

Damit die Sicherung der Daten nicht vergessen wird, kann ich einen monotone-Server in regelmäßigen Abständen sichern. Hier habe ich das Problem, dass einerseits beim Sichern kein Prozess auf das Repository zugreifen darf, andererseits der monotone-Server möglichst ständig laufen soll.

Dieses Problem löse ich indem ich ein dediziertes Backup-Repository anlege, das ich unmittelbar vor der Sicherung mit dem monotone-Server synchronisiere und dann sichere. Der Nachteil dieses Verfahrens ist, dass ich Extra-Speicherplatz für das Backup-Repository benötigt. Dafür kann der monotone-Server ununterbrochen durchlaufen. Es ist auch nicht notwendig, das Backup-Repository auf dem gleichen Rechner wie den Server zu betreiben, da beide Repositories sowieso über Netz synchronisiert werden.

Eine Alternative zu Sicherung der Server-Datenbank ohne zusätzliche Synchronisation gibt es, falls der monotone-Server über das Programm *usher* (siehe Anhang A2) gestartet wird. Usher könnte angewiesen werden, den Dienst zu stoppen und der DB-Lock würde aufgehoben werden. Für den Zeitraum

der Datensicherung würden dann Client-Prozesse zwar trotzdem nicht auf die Datenbank zugreifen können, doch usher würde ihnen über eine dedizierte Fehlermeldung die Nichtverfügbarkeit mitteilen.

Wichtig ist, die Konfiguration und Schlüssel des monotone-Servers zu sichern. Diese finden sich bei einem monotone-Server auf Debian GNU/Linux meist im Verzeichnis */etc/monotone/* und in der Datei */etc/default/monotone*. Die Schlüssel sind hierbei nicht für die Signatur wichtig, sondern für die Absicherung der Kommunikation. Wenn ich bei der Wiederherstellung einen neuen Serverschlüssel erzeugen muss, bekommen alle Beteiligten eine Meldung, dass sich der Serverschlüssel geändert hat. Dann müssen sie sich von der Authentizität des neuen Server-Schlüssels überzeugen, den alten verwerfen und den neuen akzeptieren.

Zur Wiederherstellung gehe ich vor wie beim lokalen Repository, das heißt, ich kopiere das Repository und die Konfigurationsdaten an ihren Platz und arbeite mit dem Stand der letzten Sicherung weiter.

Anpassungen und Erweiterungen

Monotone nutzt die Programmiersprache *Lua* für Anpassungen und Erweiterungen.

Lua-Funktionen werden in *rcfiles*[7] definiert, die bei jedem Lauf von monotone gelesen werden. Als *rcfiles* werden verwendet:

- *~/.monotone/monotonerc* bzw. *%APPDATA%\monotone\monotonerc* (auf MS Windows)
- *_MTN/monotonerc* im aktuellen Arbeitsbereich
- Dateien, die mit `--rcfile=Dateiname` in der Kommandozeile angegeben werden
- alle Dateien in Verzeichnissen, die mit `--rcfile=Verzeichnisname` in der Kommandozeile angegeben werden

Zuerst wird die Datei *~/.monotone/monotonerc*, dann *_MTN/monotonerc* und schließlich die mit *--rcfile* angegebenen Dateien in der Reihenfolge der Kommandozeile geladen. Spätere Definitionen überschreiben dabei frühere.

Lua-Funktionen werden auf zwei Arten in monotone verwendet. Als *Hooks* und als benutzerdefinierte Befehle. Beide Arten werden ausführlich auf der Website von monotone beschrieben.

Lua-Hooks

Hooks sind Lua-Funktionen, die von monotone an verschiedenen Stellen aufgerufen werden. Monotone stellt Default-Definitionen für einige dieser Funktionen bereit. Für andere Hooks gibt es keine Definitionen, stattdessen wird ein Default-Rückgabewert verwendet.

Für eigene Definitionen kann es hilfreich sein, den Code der alten Definition zumindest teilweise zu verwenden. Das ist mit folgendem Code möglich:

```
do
    local old_hook = default_hook
    function default_hook(arg)
```

[7] siehe Anhang A0: Rcfiles

```
            if not old_hook(arg) then
                -- do other stuff
            end
        end
    end
```

Die alte Definition von *default_hook* ist über die Variable *old_hook* nur innerhalb des *do* Blocks verfügbar. Global, das heißt, für monotone ist *default_hook* jetzt in dieser Funktion definiert.

Benutzerdefinierte Befehle

Monotone stellt eine Reihe von Hilfsfunktionen zur Verfügung, die nicht im Standard-Lua zur Verfügung stehen. Eine dieser Funktionen ist:

```
register_command(name, params, abstract, description, function)
```

Diese Funktion fügt einen Befehl *name* zu den Benutzerbefehlen von monotone hinzu. Wenn dieser registrierte Befehl aufgerufen wird, wird monotone die mit *function* bereitgestellte Definition aufrufen. Diese Funktion würde üblicherweise `mtn_automate` (siehe unten) verwenden, um den Aufruf abzuarbeiten. *params* ist eine Zeichenkette mit der Liste der Parameter für den Befehl. *abstract* ist eine kurze Beschreibung, *description* eine längere Beschreibung des Benutzerbefehls. Bei `mtn help` werden *params*, *abstract* und *description* ausgegeben.

Eine weitere wichtige Funktion ist:

```
mtn_automate(command args... )
```

Diese Funktion ruft den Befehl `automate command` von monotone mit *args* auf. Das Ergebnis ist ein *pair* (Lua), das besteht aus einem Boolean (`true` bei Erfolg) und einer Zeichenkette, die den Inhalt von `stdout` des Aufrufs von `automate command` enthält.

Diese Funktion ist nicht für den Gebrauch in normalen Lua-Hooks, sondern eher für Lua-Funktionen, die mit `register_command` registriert wurden.

Beachte, dass Tastatureingaben wie bei der `--non-interactive` Option von monotone deaktiviert sind. Aktionen, die Operationen mit passwortgeschützten Schlüsseln benötigen, werden daher nicht funktionieren, es sei denn, dass der `get_passphrase` Hook eingerichtet wurde.

Beispiel: Einbetten der aktuellen Revision

Bei einigen Versionsverwaltungssystemen - zum Beispiel RCS und CVS - gibt es Schlüsselworte, die im Quelltext eingesetzt werden können und dann vom

Versionsverwaltungssystem durch die entsprechenden Werte, wie die Version der Datei oder das Datum des Check-in, ersetzt werden. So etwas gibt es bei monotone nicht, sodass man sich hier im Bedarfsfall mit einem Script behilft, das eine Datei automatisch generiert, die dann im Quelltext eingebunden wird.

So etwas habe ich bei diesem Text ebenfalls verwendet. Die Angaben zur Basis- und aktuellen Revision werden durch folgendes Script generiert:

```
--  revision.lua -- Monotone extension command "mtn revision"

register_command(
    "revision", "",
    "Print info about actual revision.",
    "Determines the base revision and whether the current " ..
    "revision is different. Prints the information " ..
    "suitable for inclusion into restructured text.",
    "command_revision"
)

function say(abc) io.stdout:write(abc .. "\n") end

function command_revision()
    rc, txt = mtn_automate("get_base_revision_id")
    base_rev = string.match(txt,"%x+")
    if nil == base_rev then
        base_rev = ""
    end
    input, output, pid = spawn_pipe("mtn", "ls", "changed")
    res, rc = wait(pid)
    changed = output:read('*a')
    if 0 == string.len(changed) and "" ~= base_rev then
        curr_rev = base_rev
    else
        rc, txt = mtn_automate("get_current_revision_id")
        curr_rev = string.match(txt,"%x+")
    end
    say(":Autor: Mathias Weidner")
    say(":Datum: " .. os.date('%Y-%m-%d'))
    say(":Basisrevision: " .. base_rev)
    say(":Aktuelle Revision: " .. curr_rev)
    say(":Lizenz: CC BY-SA 3.0 (Creative Commons)")
end
```

Dieses Script wird im Makefile auf die folgende Art aufgerufen:

```
revision.txt: $(SOURCES) $(IMAGES) preamble.tex Makefile
        mtn --rcfile lua/revision.lua revision > revision.txt
```

Anschließend wird die Datei *revision.txt* im Vorspann eingebunden:

```
.. include:: revision.txt
```

Das Ergebnis ist auf Seite 3 dieses Textes zu sehen.
Ich gebe hier die Basisrevision und die aktuelle Revision aus, sodass man erkennen kann,

- auf welcher Basisrevision im Repository der Text aufbaut
- ob an dieser Revision Veränderungen vorgenommen wurden (dann sind Basis- und aktuelle Revision verschieden)

A0 Begriffe

Arbeitsbereich Ein Verzeichnis, in dem sich Dateien befinden, die unter Revisionskontrolle stehen, nennt man Arbeitsbereich. Im Arbeitsbereich wird der Inhalt der Dateien unter Revisionskontrolle geändert.

Monotone legt nur ein Unterverzeichnis *_MTN* in der Wurzel des Arbeitsbereiches an. Aus diesem Grund sucht monotone nach diesem Verzeichnis, für den Fall, dass es in einem anderen Unterverzeichnis aufgerufen wurde. Bevor ein Befehl ausgeführt wird, sucht monotone im aktuellen und allen übergeordneten Verzeichnissen bis zur Wurzel des Dateisystems nach einem Verzeichnis namens *_MTN*. Diese Suche kann mit der Option `--root` eingeschränkt oder mit `--no-workspace` ganz unterbunden werden.

Benutzerdefinierte Befehle Monotone kann mit eigenen Befehlen erweitert werden. Diese Befehle werden in der Script-Sprache *Lua* definiert.

Bisektion, Zweiteilung Ein Verfahren zur Fehlereingrenzung, das von monotone mit den *bisect* Befehlen unterstützt wird. Dabei werden am Anfang je eine gute und eine schlechte Revision mit Ihren IDs gekennzeichnet. Monotone teilt dann die Versionsgeschichte von der guten zur schlechten Revision jeweils etwa bei der Hälfte für den nächsten Test. Die gefundene Revision wird getestet und ebenfalls gut oder schlecht markiert. Daraufhin teilt monotone die verbleibende Historie zwischen gut und schlecht wieder für den nächsten Test.

Branch, Zweig Wenn in einer Versionsgeschichte eine *Gabelung* aufgetreten ist und man beide Enden separat weiter bearbeiten will, kann man einen neuen Zweig starten indem man einer der beiden Revisionen mit der Option `--branch` einen neuen Zweignamen zuweist.

Branchname, Zweigname Der Name eines Zweiges sollte global eindeutig sein, da es ansonsten, wenn das Repository mit anderen geteilt wird, zu Kollisionen im Namensraum kommt. Zwei verschiedene Projekte mit dem gleichen Namen können nicht in einer Datenbank gespeichert werden.

Als Standard für die Namensvergabe wird empfohlen, den eigenen DNS-Namen voranzustellen (zum Beispiel *net.mamawe.text.monotone-brevier*

für diesen Text). Man fängt dabei mit dem letzten Domain-Bestandteil an und fügt je nach Bedarf weitere hinzu.

Datenbank Monotone verwendet als lokales Repository eine SQLite-Datenbank. Daher werden die Begriffe lokales Repository und Datenbank synonym verwendet.

DVCS Distributed Version Control System, verteilte Versionsverwaltung. Es gibt kein zentrales Repository, welches die Änderungen verfolgt, sondern mehrere, die untereinander synchronisiert werden.

Fork, Gabelung Im Laufe der Entwicklung folgt normalerweise eine *Revision* auf die andere, wie die Perlen an einer Kette. Nun ist es möglich, dass ausgehend von einer *Revision* (zum Beispiel durch verschiedene Entwickler) verschiedene Änderungen in einem Verzeichnisbaum oder einer Datei gemacht wurden. An dieser Stelle gabelt sich die Revisionsgeschichte und wir haben mehrere letzte Revisionen. Diesen Vorgang nennt man *Fork* bzw. *Gabelung*.

Hook *Hooks* sind *Lua* Funktionen, die von monotone an verschiedenen Stellen aufgerufen werden. Monotone stellt für einige dieser Funktionen Default-Definitionen bereit, für andere liefert es Default-Rückgabewerte. *Hooks* können in *rcfiles* durch eigene Definitionen ersetzt werden.

Keystore Monotone arbeitet sehr viel mit Zertifikaten, die mit asymmetrischen Schlüsseln signiert werden. Während die öffentlichen Schlüssel in der Datenbank abgelegt und verteilt werden, liegen die privaten Schlüssel als Dateien unter UNIX im Verzeichnis *$HOME/.monotone/keys* bzw. unter MS Windows in *%APPDATA%\monotone\keys*. Dieses Verzeichnis nennt man den Keystore.

Lua *Lua* ist eine leichtgewichtige Script-Sprache, die in andere Anwendungen eingebaut werden kann. Monotone enthält einen Interpreter für *Lua*, der für *Hooks* und *benutzerdefinierte Befehle* verwendet wird.

Manifest, Roster Manifeste sind interne textbasierte Datenstrukturen, die alle Dateien, Verzeichnisse sowie deren Attribute für eine Revision erfasst.

Seit Version 0.26 heißen diese Datenstrukturen *Roster* und enthalten noch mehr Informationen über die Dateien.

Merge, Zusammenführen Wenn in der Versionsgeschichte ein Fork aufgetreten ist, und die jeweils letzten Revisionen wieder zusammengeführt werden sollen, nennt man das bei Versionsverwaltungssystemen *Merge*. Monotone führt dabei ein Paar von 3-Wege-Zusammenführungen aus. Zum

einen auf Verzeichnisebene um Differenzen im Verzeichnisbaum (zum Beispiel Umbenennungen) aufzulösen, zum anderen zeilenweise für jede geänderte Datei um Unterschiede durch gleichzeitiges Bearbeiten der gleichen Datei aufzulösen.

Die 3-Wege-Zusammenführung ist nicht einfach nur ein Anwenden von Änderungen der einen Seite auf die andere. Zunächst wird der nächste gemeinsame Vorgänger beider Versionen in der Versionsgeschichte gesucht. Dann werden die Änderungen der linken und der rechten Seite von diesem Vorgänger berechnet und die Änderungskoordinaten basierend auf den Änderungen der linken Seite berechnet. Erst dann werden die beiden Versionen verknüpft, wobei identische Änderungen ignoriert werden und widersprechende Änderungen zurückgewiesen werden. Falls es widersprechende Änderungen gab, werden diese an eine (Lua-) Funktion übergeben, deren Standardimplementation den Editor Emacs im Diff-Modus aufruft.

Monotone unterscheidet den *Merge* innerhalb eines Zweiges, für den es den Befehl `mtn merge` gibt und den *Merge* von einem Zweig zu einem anderen, für den der Befehl `mtn propagate` verwendet wird.

Rcfiles Monotone kann durch eigene Funktionen in der Script-Sprache *Lua* angepasst und erweitert werden. Diese Funktionen werden in sogenannten *rcfiles* abgelegt, welche bei jedem Start von monotone eingelesen werden.

Revision Eine Revision umfasst alle Änderungen, die von einer *Version* zur nächsten führen. Die Revision wird als interne textbasierte Datenstruktur in monotone gepflegt. Siehe auch *Manifest*, *Versionsgeschichte*.

Roster Siehe *Manifest*

Schlüssel Monotone verwendet für die Integritätsprüfung Zertifikate, die mit asymmetrischen Schlüsseln der Entwickler signiert sind. Die öffentlichen Schlüssel der Beteiligten werden dabei in den einzelnen Datenbanken mit abgelegt, die privaten Schlüssel werden im *Keystore* des Benutzers abgelegt.

Als Verschlüsselungsverfahren verwendet monotone *RSA*.

In älteren monotone-Versionen wurden Schlüssel über ihrem Namen (der E-Mail-Adresse) identifiziert. Es konnte immer nur ein Schlüssel mit einem Namen in einer Datenbank vorhanden sein. In neueren Versionen werden Schlüssel über ihren SHA1-Hash identifiziert. Damit ist es nun möglich, mehrere Schlüsse mit demselben Namen in einer Datenbank zu haben. Das ist insbesondere dann praktisch, wenn ein (privater) Schlüssel

verloren gegangen ist. Über einen Lua-Hook können Schlüsseln lokale Namen vergeben werden, die anstelle des Namens des Schlüssels angezeigt werden. Damit kann man unterschiedliche Schlüssel mit dem gleichen Namen unterscheiden.

Server Ein monotone-Prozess, der über das Netzwerk angesprochen werden kann. Das kann ein ständig laufender monotone-Server sein oder eine kurzzeitig mit `mtn serve` zum Abgleich bereitgestellte Datenbank auf einer Arbeitsstation.

SHA1-Hash In monotone wird die *SHA1* Funktion verwendet, um eindeutige Bezeichner für bestimmte Versionen von Dateien zu bestimmen. Diese Bezeichner sind 20 Byte lang und werden SHA1-Hash genannt.

Diese Hash wird meist als Folge von 40 Hexadezimalzahlen lesbar ausgegeben und auch so eingegeben, wenn ein Befehl die Angabe einer Hash erfordert. Dabei müssen nur soviele Stellen angegeben werden, wie nötig sind, um die Hash eindeutig zu identifizieren.

Standarddatenbank In älteren Versionen von monotone war es notwendig, für alle Operationen, die mit dem lokalen Repository zu tun hatten, den Pfad zur Datenbankdatei mit der Option *--db* anzugeben, wenn man sich nicht gerade in einem Arbeitsbereich befand. Bei neueren Versionen von monotone gibt es die Konvention, dass die Datenbank sich im Verzeichnis *$HOME/.monotone/databases/* befindet, wenn dem Namen ein Doppelpunkt vorangestellt wird. Wenn man bei diesen Versionen von monotone die Angabe der Datenbankdatei weglässt, so verwendet monotone *:default.mtn* als Datenbank. Diese Datei nennt man darum auch Standarddatenbank.

URI Uniform Resource Identifier ist ein einheitlicher Bezeichner für Ressourcen, mit dem diese unabhängig vom verwendeten Programm oder Betriebssystem benannt werden können. Ein URI besteht aus einem Schema, gefolgt von einem Doppelpunkt gefolgt von den näheren Angaben entsprechend der Schreibweise für das Schema.

Monotone versteht von Haus aus die Schemata *ssh* und *file* für die Synchronisation mit anderen Datenbanken.

Version Bei monotone wird unterschieden zwischen Versionen von Dateien und Versionen von Verzeichnisbäumen. Generell ist eine Version der Zustand einer Datei oder eines Verzeichnisbaums zwischen zwei Änderungen. Die Version vor einer Änderung wird Elternversion, die Version nach dieser Änderung wird Kindesversion genannt.

Eine Dateiversion bezieht sich auf den Inhalt der Datei. Um die Dateiversion eindeutig zu referenzieren, wird mit der *SHA1* Funktion eine Folge von 20 Bytes (genannt Hash) erzeugt, die genau dieser speziellen Version zugeordnet ist.

Eine Version eines Verzeichnisbaumes wird gebildet, indem zunächst das *Manifest* erzeugt wird. Dieses enthält eine Liste aller Dateiversionen von Dateien im Verzeichnisbaum und gegebenenfalls weitere Attribute dieser Dateien. Vom *Manifest* wird ebenfalls eine SHA1-Hash gebildet.

Versionsgeschichte Die verschiedenen Versionen von Dateien und Verzeichnisbäumen selbst haben keinen Bezug zueinander. Das heißt, man kann aus den SHA1-Hashes und den Manifesten nicht erkennen, welche Version von welcher anderen abgeleitet wurde. Um die komplette Versionsgeschichte zu erhalten, wird daher für jede Änderung eine *Revision* gepflegt. Diese enthält das neue Manifest, die vorherige Revision, an die die aktuelle anschließt und welche Dateiänderungen im Einzelnen vorgenommen wurden. Die *Revision* selbst ist auch eine Textdatei und wird deshalb ebenfalls über eine SHA1-Hash identifiziert.

Zertifikat Jede Revision in monotone wird über eine SHA1-Hash identifiziert, die sowohl den Inhalt der Revision als auch ihre ganze Abstammung repräsentiert. Will man weitere Aussagen zu einer Revision (zum Beispiel den Zeitpunkt der Erstellung der Revision) treffen, so sind diese zunächst unabhängig von der Revision und leicht änderbar. Um diese zusätzlichen Aussagen zu sichern, fügt monotone einen Schlüssel und eine mit diesem Schlüssel erzeugte Signatur über die zusätzlichen Aussagen an. Diese 4 Elemente (die Revision, die zusätzliche Information, der Schlüssel und die Signatur) zusammen bilden ein Zertifikat.

Monotone nutzt Zertifikate ausgiebig. Sämtliche zusätzliche Information, die gespeichert, übertragen oder aufgerufen werden soll, wird in Form von Zertifikaten verwendet.

Zweig Siehe *Branch*.

Zweiteilung Siehe *Bisektion*

A1 Weiterführende Informationen

E-Mail, Chat

E-Mail http://lists.nongnu.org/mailman/listinfo/monotone-users

IRC irc.oftc.net #monotone

World Wide Web

www.monotone.ca Die Homepage von monotone (englisch), hier findet man fast alles, was man braucht.

wiki.monotone.ca Das Wiki zu monotone. Von dort habe ich sehr viele Anregungen für diesen Text genommen. Einzelne Passagen sind direkte Übersetzungen von Wiki-Artikeln oder Teilen davon.

guitone.thomaskeller.biz Die Homepage von Guitone, einem grafischen Frontend für monotone, das insbesondere Anfängern den Einstieg erleichtern dürfte.

diffuse.sourceforge.net Die Homepage von diffuse, einem Programm zum Vergleichen und Zusammenführen von Dateien.

kdiff3.sourceforge.net Die Homepage von KDiff3, einem Programm zum Vergleichen und Zusammenführen von Dateien.

www.lua.org Die Homepage zur Programmiersprache *Lua*, mit der Anpassungen und Erweiterungen für monotone geschrieben werden können.

furius.ca/xxdiff Die Homepage von xxdiff, einem Programm zum Vergleichen und Zusammenführen von Dateien.

oandrieu.nerim.net/monotone-viz Die Homepage von monotone-viz, einem Visualisierungsprogramm für Revisionsgraphen.

nakedstartup.com/2010/04/simple-daily-git-workflow Eine Beschreibung für einen möglichen Arbeitsablauf mit *git*, die ich als Anregung für diesen Text verarbeitet habe.

www.duden.de Auf den Webseiten von *Duden online* gibt es die Möglichkeit, Rechtschreibung und Grammatik überprüfen zu lassen. Davon habe ich ausgiebig Gebrauch gemacht. Alle im Text verbliebenen Fehler gehen auf mich.

www.troyhunt.com/2011/05/10-commandments-of-good-source-control.html *The 10 commandments of good source control management* (englisch)

Das Epigraph über dem Grundlagenkapitel ist von dieser Seite genommen. Auch die meisten der anderen Gebote gehören meiner Meinung nach zu den Grundprinzipien der Arbeit mit Versionsverwaltungssystemen.

www.mail-archive.com/monotone-devel@nongnu.org/msg09319.html Posting auf der E-Mail-Liste *monotone-devel*

Re: [Monotone-devel] mtn automate get_current_revision_id - bug or feature?, *Nuno Lucas*

Ein Python-Script, das ich als Vorlage für die Ermittlung von Basisrevision und aktueller Revision, wie im Vorspann angezeigt, genommen habe.

weidner.in-bad-schmiedeberg.de/computer/rcs/monotone Artikel auf meiner Homepage zu monotone, zum Teil Auszüge aus und Vorabversionen von diesem Text.

A2 Grafische Oberflächen und andere Programme

diffuse

Diffuse ist ein kleines und einfaches Werkzeug zum Zusammenführen von Texten, geschrieben in Python. Mit Diffuse können Sie ganz einfach Änderungen in Ihren Quelltexten zusammenführen, bearbeiten und prüfen. Diffuse ist freie Software.

Die Homepage von diffuse ist http://diffuse.sourceforge.net/.

Das Programm läuft auf allen Betriebssystemen, auf denen *Python* verfügbar ist (das heißt: UNIX, Mac OS, MS Windows).

Guitone

Guitone ist eine grafische Benutzeroberfläche für monotone die insbesondere für Anfänger gedacht ist.

Es gibt einen Installer für Microsoft Windows. Die Installation unter Linux ist relativ einfach (Version 1.0rc5 auf Ubuntu):

```
Holen der Quellen von http://guitone.thomaskeller.biz

$ sudo aptitude install qt4-make libqt4-dev
$ tar xzf guitone-1.0rc5.tgz
$ cd guitone-1.0rc5
$ qmake guitone.pro
$ make
$ sudo mkdir -p /usr/local/stow/guitone-1.0rc5/bin
$ sudo cp bin/guitone /usr/local/stow/guitone-1.0rc5/bin/
$ sudo stow -d /usr/local/stow -v guitone-1.0rc5
$ guitone
```

Indefero

Indefero ist im Grunde ein Klon von GoogleCode mit Support für verschiedene DVCS und seit Version 1.1 auch für monotone. Dieses scheint zurzeit (2011) aktiver gepflegt zu werden als *viewmtn*.

Die Homepage der Open Source Version ist http://www.indefero.net/open-source/.

Wichtig! Auf der Homepage wird zur Zeit Version 1.0 zum Download angepriesen. Unterstützung für Monotone gibt es aber erst für Versionen ab 1.1, die es auf der Download-Seite auch gibt.

KDiff3

KDiff3 ist ein Vergleichs- und Zusammenführungs-Programm wie *diffuse* und *xxdiff*.

Die Homepage von KDiff3 ist http://kdiff3.sourceforge.net/.

Es gibt Versionen für UNIX, Mac OS X, MS Windows.

Monotone::AutomateStdio

Monotone::AutomateStdio ist ein Perl-Modul zum Zugriff auf das *automate stdio* Interface von monotone.

Die Homepage ist http://www.coosoft.plus.com/software.html.

Monotone::AutomateStdio ist via CPAN, das Comprehensive Perl Archive Network (http://search.cpan.org/~aecooper/Monotone-AutomateStdio/), zu bekommen.

Monotone-viz

Monotone-viz ist ein Programm zur Visualisierung von Versionsgraphen. Ein Teil der Bilder in diesem Text ist mit monotone-viz erzeugt. Das Programm benötigt `dot` von *GraphViz* und `Gtk+`.

Die Homepage von monotone-viz ist http://oandrieu.nerim.net/monotone-viz/.

Bei Debian/Ubuntu kann man monotone-viz mit der Softwareverwaltung (*apt-get*) installieren. Für Windows gibt es ein Zip-Archiv und eine Installationsanleitung.

usher

Usher ist wie der Name besagt, eine Art Pförtnerprogramm für monotone-Server. Es fragt den monotone-Client, was er synchronisieren will, sucht den zuständigen Server in einer Tabelle und leitet die Verbindung dann weiter.

Die Homepage von usher ist http://mtn-host.prjek.net/projects/webhost/.

Dort ist zurzeit ein Archiv für Version 0.99 zu finden, dem das Verzeichnis mit der Dokumentation fehlt. Am besten synchronisiert man ein lokales Repository mit dem Projekt:

```
$ mtn --db :usher.mtn clone webhost.mtn-host.prjek.net \
      net.venge.monotone.contrib.usher usher
$ cd usher
$ lynx usher/doc/documentation.html
```

Dann ist es auch einfacher, eigene Änderungen zurückzugeben.

viewmtn

Viewmtn ist ein Web-Interface für monotone. Unter http://mtn-view.1erlei.de/ kann man eine öffentliche Installation finden, um sich einen Eindruck davon zu verschaffen.

Die Homepage von viewmtn ist http://viewmtn.1erlei.de/.

Viewmtn ist in Python geschrieben und nutzt das web.py-Framework. In der Datei INSTALL ist beschrieben, wie man es aufsetzen kann.

xxdiff

Xxdiff ist ein grafisches Vergleichs- und Merge-Programm für Dateien und Verzeichnissen. Dieses Programm wird in der monotone-FAQ als das beste bekannte externe Merge-Programm angegeben.

Die Homepage von xxdiff ist http://furius.ca/xxdiff/.

Bei Debian/Ubuntu kann man xxdiff mit der Softwareverwaltung (*apt-get*) installieren.

Für Windows gibt es eine Version über die Homepage. Diese Version hat einige Einschränkungen:

> There are some problems with the Windows binary, it crashes when you display the options dialog. But other than that it seems most is working.

Hier ist dann möglicherweise eines der anderen Merge-Programme vorzuziehen, wie z.B. *diffuse* oder *KDiff3*.

A3 Arbeitsablauf Spickzettel

Dieser Arbeitsablauf geht von einem eigenen Zweig pro Fehlerbeseitigung / Ticket aus.

Neuen Zweig auschecken

```
$ mtn --db ticketnr.mtn db init
$ mtn --db ticketnr.mtn pull remotehost --branch trunk
```

Umstellen auf den neuen Zweig:

```
$ mtn --db ticketnr.mtn co --branch trunk .
$ mtn cert h: branch trunk.ticketnr
$ mtn update --branch trunk.ticketnr
```

Arbeit am Problem

```
$ mtn list unknown
$ mtn add | move | remove
$ mtn status | diff
$ mtn commit -m "kurze Beschreibung"
```

Änderungen veröffentlichen

```
$ mtn pull
$ mtn show_conflicts | conflicts
$ mtn propagate trunk trunk.ticketnr

$ mtn propagate trunk.ticketnr trunk
$ mtn sync
```

Falls jemand anders inzwischen Änderungen veröffentlicht hat:

```
$ mtn show_conflicts | conflicts
$ mtn merge trunk
$ mtn sync
```

A4 CVS Spickzettel

Dieser Anhang ist dem *CVS Phrase Book* aus dem monotone-Wiki entnommen. Er soll insbesondere denen helfen, die bereits mit dem Versionsverwaltungssystem CVS gearbeitet haben.

Einen Zweig auschecken

CVS:

```
$ CVSROOT=:pserver:cvs.example.net/wobbler
$ cvs -d $CVSROOT checkout -r 1.2
```

monotone:

```
$ mtn --db=/path/to/database.mtn \
      pull "mtn://monotone.example.net?net.example.wobbler*"
$ mtn --db=/path/to/database.mtn \
      checkout --revision=fe37 wobbler
```

Vor dem Checkout ist üblicherweise kein Arbeitsbereich vorhanden, sodass man die Datenbank bei bei älteren Versinen von monotone benennen musste. Beim Aufruf von monotone in einem Arbeitsbereich war das nicht notwendig. Bei neueren Versionen von monotone wird ohne explizite Angabe der Datenbank die Standarddatenbank[8] verwendet.

Änderungen übergeben

CVS:

```
$ cvs commit -m "log message"
```

monotone:

```
$ mtn commit --message="log message"
$ mtn push "mtn://monotone.example.net?net.example.wobbler*"
```

[8] siehe Anhang A0: Standarddatenbank

Änderungen zurücknehmen

CVS:

```
$ cvs update -C file
```

monotone:

```
$ mtn revert file
```

Monotone hat einen separaten *revert* Befehl, um lokale Änderungen zurückzunehmen und auf den Stand der Basis-Revision zu bringen. Monotone besteht auf die Angabe einer Datei oder eines Verzeichnisses. Um alles zurückzusetzen, gibt man `mtn revert .` ein.

Mit dem *revert* Befehl werden auch gelöschte Dateien wiederhergestellt. Dafür gibt es die praktische Option `--missing`, die genau die fehlenden Dateien benennt.

Andere Änderungen importieren

CVS:

```
$ cvs update -d
```

monotone:

```
$ mtn pull "mtn://monotone.example.net?net.example.wobbler*"
$ mtn merge
$ mtn update
```

Bei monotone ist die Synchronisierung von der Aktualisierung getrennt. Der *merge* Befehl ist nur notwendig, wenn man am neuesten Stand des gesamten Revisions-Zweiges interessiert ist. Will man nur Änderungen an der eigenen Basis-Revision verfolgen, kann man diesen weglassen. Erst der *update* Befehl aktualisiert den eigenen Arbeitsbereich.

Revisionen bezeichnen

CVS:

```
$ cvs tag FOO_TAG .
```

monotone:

```
$ mtn tag h: FOO_TAG
```

Bei CVS werden Bezeichner (Tags) immer an einzelne Dateien vergeben und man erkennt eine über das ganze Repository konsistente Revision am ehesten am gleichen Bezeichner der Dateiversionen. Demgegenüber werden alle Änderungen in monotone immer über das ganze Repository erfasst. Von diesen Änderungen sind dann einige mit Bezeichnern versehen. Es gibt bei monotone keine Bezeichner nur für einen Teil der Dateien im Repository.

Arbeitsbereich auf eine andere Revision bringen

CVS:

```
$ cvs update -r FOO_TAG -d
```

monotone:

```
$ mtn update -r 830ac1a5f033825ab364f911608ec294fe37f7bc
```

oder

```
$ mtn update -r t:FOO_TAG
```

Mit dem Revisions-Parameter verhält sich monotone ähnlich wie CVS. Ein Unterschied ist, dass man direkt von der gewählten Revision aus weiterarbeiten und mit *commit* einchecken kann. Das ist insbesondere dann nützlich, wenn man nach einer Aktualisierung feststellt, dass die Endversion eines Zweiges defekt ist. Dann kann man auf eine ältere, funktionierende Funktion zurückgehen und normal weiterarbeiten. Später kann man die eigene Arbeit dann mit der wieder funktionierenden Endversion zusammenführen.

Unterschiede ansehen

CVS:

```
$ cvs diff
$ cvs diff -r 1.2 -r 1.4 myfile
```

monotone:

```
$ mtn diff
$ mtn diff -r 3e7db -r 278df myfile
```

Status des Arbeitsbereiches

CVS:

```
$ cvs status
```

monotone:

```
$ mtn status
```

Verzeichnisse und Dateien hinzufügen

CVS:

```
$ cvs add dir
$ cvs add dir/subdir
$ cvs add dir/subdir/file
```

monotone:

```
$ mtn add dir/subdir/file
```

Um eine Datei hinzuzufügen, reicht es, den kompletten Pfad anzugeben. Verzeichnisse werden, wenn nötig, automatisch mit aufgenommen.

Verzeichnisse und Dateien entfernen

CVS:

```
$ rm file.txt
$ cvs remove file
```

monotone:

```
$ mtn drop file
```

Mit dem monotone-Befehl *drop* wird die Datei sowohl aus dem Dateisystem als auch aus dem Manifest entfernt.

Historie ansehen

CVS:

```
$ cvs log [file]
```

monotone:

```
$ mtn log [file]
```

Mit den Optionen `--from` und `--to` kann eingeschränkt werden, zwischen welchen Revisionen die Historie ausgegeben werden soll.

Ein neues Projekt importieren

CVS:

```
$ cvs import wobbler vendor start
```

monotone:

```
$ mtn --db=/path/to/database.mtn --branch=net.example.wobbler \
      setup .
$ mtn add -R .
$ mtn commit
```

Der *setup* Befehl verwandelt ein gewöhnliches Verzeichnis in einen monotone-Arbeitsbereich. Danach kann man die Dateien hinzufügen und mit *commit* übergeben.

Ein Repository initialisieren

CVS:

```
$ cvs init -d /path/to/repository
```

monotone:

```
$ mtn db init --db=/path/to/database.mtn
```

Monotones lokales Repository ist eine einzelne Datenbankdatei, die nur von mir benutzt wird. Diese Datei kann irgendwo angelegt werden und braucht keine Lese- oder Schreibrechte für andere.

Kolophon

Dieses Buch ist mit Hilfe der Python Docutils entstanden. Die Druckversion wurde mit *rst2latex* in eine LaTeX-Eingabedatei übersetzt und anschließend mit *pdflatex* in das PDF-Format umgewandelt.

Natürlich handelt das Buch nicht nur von *monotone*, der Text steht selbst unter Versionsverwaltung. Der Docutils-Quelltext für Seite 3 mit den Versionsinformationen ist direkt aus Monotone generiert. Wie, ist im Kapitel *Anpassungen und Erweiterungen* beschrieben.

Die Maus auf dem Umschlag wurde ursprünglich von Graydon Hoare, dem initialen Autor von monotone, von einer Mausfotografie mit Inkscape in SVG gezeichnet und dann später durch Thomas Keller etwas nachbearbeitet.

Den Umschlag selbst habe ich mit Hilfe eines Templates von Lulu und dem Programm Gimp erstellt.