mehr zum thema:
mercurial.selenic.com

„MERCURIAL”: SCHNELLE UND SKALIERBARE VERSIONSVERWALTUNG

Verteilte Versionsverwaltung wird heutzutage von vielen Firmen und Open-Source-Projekten eingesetzt. In diesem Artikel werfen wir einen genaueren Blick auf Mercurial, einem der schnellsten und benutzerfreundlichsten Vertreter der „Distributed Revision Control Systems”. Wir erklären, wie dieses Tool funktioniert und warum man es einsetzen sollte.

die autoren

Dr. Martin Geisler
(E-Mail: mg@aragost.com)
ist einer der Hauptentwickler von Mercurial. Er lebt in Zürich, wo er für die aragost Trifork AG als Berater arbeitet und anderen Firmen hilft, Mercurial einzusetzen.

Aaron Digulla
(E-Mail: digulla@hepe.com)
ist User von Mercurial und arbeitet als Senior Java Developer bei Avanon in Zürich.

Klaus Bucka-Lassen
(E-Mail: kbl@aragost.com)
ist User von Mercurial und Geschäftsführer der aragost Trifork AG. Er lebt in der Schweiz, wo er hauptsächlich als Technologieberater, Projektleiter, Architekt und Softwareentwickler für Großkunden tätig ist.

Genauso wie Subversion viele Probleme von *Concurrent Versions System (CVS)* gelöst hat, lösen verteilte Versionsverwaltungssysteme (*Distributed Revision Control Systems, DVCS*) viele Probleme von Subversion oder ähnlichen Tools. Die neue Generation von Werkzeugen ist schneller und flexibler, macht die Arbeit mit Verzweigen und Zusammenführen (*Branching* und *Merging*) deutlich einfacher und gibt den Anwendern mehr Möglichkeiten, um den Entwicklungsprozess zu gestalten.

Die bisherigen zentralisierten Werkzeuge, beispielsweise „CVS”, „Subversion”, „IBM ClearCase” und „MS Team Foundation Server”, sind zwar leistungsfähig, zwingen dem Anwender aber auch eine bestimmte Vorgehensweise auf. Statt den Entwickler zu unterstützen, werden sie häufig als Behinderung empfunden.

Verteilte Versionsverwaltungssysteme, beispielsweise „Mercurial”, „Git” und „Bazaar”, bieten von Entwicklern seit Jahren geforderte Features. In diesem Artikel wollen wir die wichtigsten davon vorstellen.

Mercurial

Als Basis in diesem Artikel verwenden wir Mercurial, eine für alle Plattformen verfügbare *Open-Source-Software (OSS)*. Mercurial wird seit einiger Zeit bereits von großen Organisationen – z. B. der Mozilla Foundation („Firefox”, „Thunderbird” usw.) und Oracle („OpenSolaris”, „Java”, „OpenOffice” usw.) – verwendet. Seit 2009 bietet Google in seinem OSS-Hosting-Projekt „Google Code” ebenfalls Mercurial an (vgl. [Goo10-a]).

Wie bei allen DVCS ist das wichtigste Feature von Mercurial, dass jeder Anwender eine komplette Kopie der gesamten Historie des Projekts auf seinem Rechner hat. Statt eines zentralen Servers gibt es nun viele redundante Systeme, die alle gleichzeitig Client und Server sind.

Durch die Verteilung der Historie werden viele Operationen extrem schnell. So dauert ein *Commit* bei Mercurial oft nur den Bruchteil einer Sekunde. Da jeder Entwickler einen kompletten Server auf seinem Rechner hat, kann man Änderungen lokal speichern, ohne Kollegen zu stören. Erst wenn man seine Änderungen veröffentlicht, werden diese für das Team sichtbar. Mit der Zeit beginnen die Entwickler, Änderungen von allein in kleinen Portionen einzuchecken, statt am Abend hunderte von Änderungen auf einmal.

Dadurch bleiben die Änderungen immer überschaubar, auch zeitlich. Wenn ein Entwickler feststellt, dass er sich in eine Sackgasse programmiert hat, kann er – ohne lange nachzudenken – die Arbeit der letzten halben Stunde verwerfen.

Schlüsselkonzepte

Wer mit der zentralisierten Versionsverwaltung vertraut ist, wird viele der dort verwendeten Konzepte bei Mercurial wiedererkennen. Auf die neuen Konzepte einer verteilten Versionsverwaltung bzw. im speziellen von Mercurial möchten wir im Folgenden näher eingehen. Wer diese Konzepte versteht, kann viel effizienter mit Mercurial arbeiten.

Arbeiten mit einem einzelnen Repository

Beginnen wir mit der Arbeitskopie (*Working Copy*). Hier befinden sich die Dateien Ihres Projekts und können verändert und übersetzt werden. Wird ein Commit ausgeführt, werden alle Änderungen aus der Arbeitskopie als neuer Änderungssatz (*Changeset*) gespeichert. Ein Änderungssatz entspricht dem Unterschied zweier Revisionen in SVN-ähnlichen Werkzeugen. Alle Änderungssätze zusammen bilden die Historie des Projekts und befinden sich im Repository, das immer im Verzeichnis .hg/ im Wurzelverzeichnis der Arbeitskopie liegt. Mercurial legt also nicht in jedem Verzeichnis Daten von sich ab, sondern nur in einem. Damit kann es beim Verschieben von Dateien in der Arbeitskopie keine Probleme mit der Versionsverwaltung mehr geben.

Die Arbeitskopie enthält den Checkout einer Revision, der so genannten *Working Copy Parent Revision*. Wird der Dateistatus verlangt, werden die Dateien in der Arbeitskopie mit dem Zustand in der *Parent Revision* verglichen. Bei einem Commit ist der neue Änderungssatz immer eine *Child Revision*, die auf die aktuelle *Working Copy Parent Revision* folgt. Jeder Änderungssatz enthält daher die Kennung seiner *Parent Revision*. Dabei handelt es

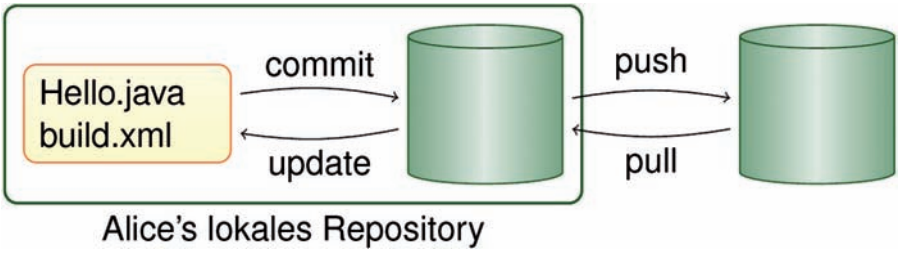

Abb. 1: Lokale und entfernte Operationen.

sich um eine lokale Operation (siehe Abbildung 1), d. h. die Arbeit kann jederzeit zwischengespeichert werden, ohne dass andere betroffen sind.

Arbeiten mit mehreren Repositorys

Repositorys können geklont werden – damit kann eine vollständige, unabhängige Kopie der gesamten Historie erstellt werden. Das ist immer der erste Schritt, wenn ein Entwickler zum Projekt stößt. Er klont sich eine Kopie von einer bestimmten Quelle, wie etwa vom Hauptserver der Firma oder von der Homepage eines OSS-Projekts. Nachdem er seinen privaten Klon erzeugt hat, kann er durch die Historie wandern und an beliebigen Stellen neue Änderungssätze erzeugen, ohne noch einmal mit dem Server zu „sprechen", von dem er seinen Klon erzeugt hat. Wenn neue Änderungssätze auf dem Server erscheinen, kann er sie mittels pull in seinen Klon holen bzw. ziehen (siehe Abbildung 1). Die gegenteilige Operation heißt push: Damit schiebt er seine neuen Änderungssätze zurück zum Server. Technisch gesehen vergleichen die beiden Befehle zwei Repositorys und übertragen fehlende Änderungssätze in die gewünschte Richtung.

Wenn Änderungssätze zwischen zwei Repositorys kopiert werden, wird die *Parent Revision* verwendet, um diese an der richtigen Stelle in der Historie des Projekts aufzuhängen (siehe Abbildung 2).

Werden Änderungssätze zwischen Repositorys verschoben, erscheinen Zweige (*Branches*) in der Historie. Sie repräsentieren die parallelen Arbeiten am Projekt. Nehmen wir an, dass Alice zum Zeitpunkt T_1 einen neuen Änderungssatz *A* erzeugt, während Bob einen Änderungssatz *B* anlegt. Beide stammen vom Änderungssatz *X* ab. Im Moment T_2 haben die Änderungssätze *A* und *B* noch keine Kinder und werden als Heads bezeichnet. Zwei Heads repräsentieren unabhängige Arbeiten am Projekt und werden zu einem geeigneten Zeitpunkt in einem Änderungssatz *M* vereint (*Merging*). Wenn Alice sich nun die Änderungen von Bob holt, sieht sie zwei Zweige, die bei *X* beginnen und die sie nun zusammenführen muss.

Änderungen zusammenführen

Bei den zentralisierten Werkzeugen sind *Merges* immer heikel. Entwickler haben bei dieser Arbeit oft ein schlechtes Gefühl, denn sie wissen aus Erfahrung, dass häufig etwas schiefgeht. Mercurial und Git wurden ursprünglich für OSS-Projekte entwickelt, also Projekte, bei denen Änderungen von tausenden von Entwicklern pro Monat zusammengeführt werden müssen. Linus Torvalds etwa muss jeden Monat bis zu 6.000 Änderungssätze zusammenführen.

Wenn es nur einen Server gibt, kann man seine Arbeit vor dem Merge nirgendwo speichern. Ein Merge mischt also immer die Änderungen von anderen Entwickler in die eigene, ungesicherte Arbeit. Bei einem Fehler ist es meistens mit einem hohen Aufwand verbunden, diesen Schritt wieder rückgängig zu machen. Mercurial hingegen führt nur Änderungssätze vom Server zusammen. Der Entwickler muss also zuerst seine Änderungen sichern und kann dann einen Merge durchführen.

Wie läuft nun ein Merge technisch ab? Im trivialen Fall existiert eine geänderte Datei nur in einem Zweig oder sie wurde nur in einem Zweig geändert. Dann muss Mercurial sie nur übernehmen. Wenn verschiedene Versionen einer Datei zusammengeführt werden sollen, sucht Mercurial erst einmal den jüngsten gemeinsamen Vorfahren. Dann wendet es alle Änderungen der Reihe nach auf diesen Vorfahren an. Das funktioniert, solange es bei den Änderungen keine Überlappung gibt. Bei einer Überschneidung startet Mercurial ein frei konfigurierbares, externes Drei-Wege-Merge-Tool. In diesem Tool kann der Entwickler die drei Versionen vergleichen und entscheiden, was zu tun ist.

Mercurial wählt das Tool auf Basis des Dateityps. Bei Textdateien kommt in der Regel „KDiff3" zum Einsatz (siehe Abbildung 3). In dem gezeigten Beispiel wurde eine Liste von Schweizer Kantonen bearbeitet. Links sieht man den gemeinsamen Vorfahren, der nur die deutschen Namen enthält. In der Mitte wird die Version aus dem einen Zweig angezeigt. Hier hat ein Entwickler die italienischen Namen hinzugefügt. Rechts ist die Version der Datei zu sehen, wie sie im anderen Zweig aussieht. Offensichtlich hat hier ein anderer Entwickler gleichzeitig die französischen Namen eingegeben. Wie gut zu erkennen ist, kann der Merge Daten aus beiden Dateien enthalten. In diesem Fall enthält das Ergebnis die Namen der Kantone in drei Sprachen.

Kommandozeile

Von der Kommandozeile wird Mercurial mit hg aufgerufen (nach dem Kürzel für das chemische Element Quecksilber, auf englisch „mercury"). Genauso wie svn bei Subversion wird auch hg mit Unterkommandos gesteuert, die in Kasten 1 aufgelistet sind. Viele dieser Kommandos gleichen denen von Subversion: Bei der Entwicklung von Mercurial wurde darauf geachtet, den Wechsel von Subversion möglichst einfach zu machen.

Nachteile zentralisierter Systeme

Nicht verteilte Systeme verwenden eine Client/Server-Architektur mit einem zentralen

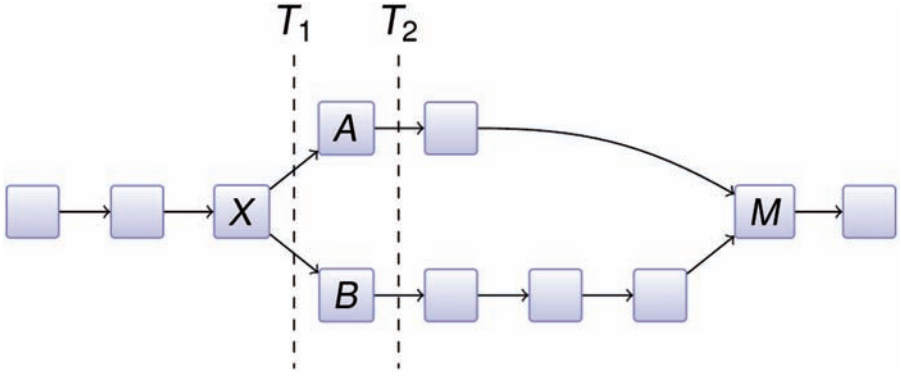

Abb. 2: Der Graph der Änderungssätze.

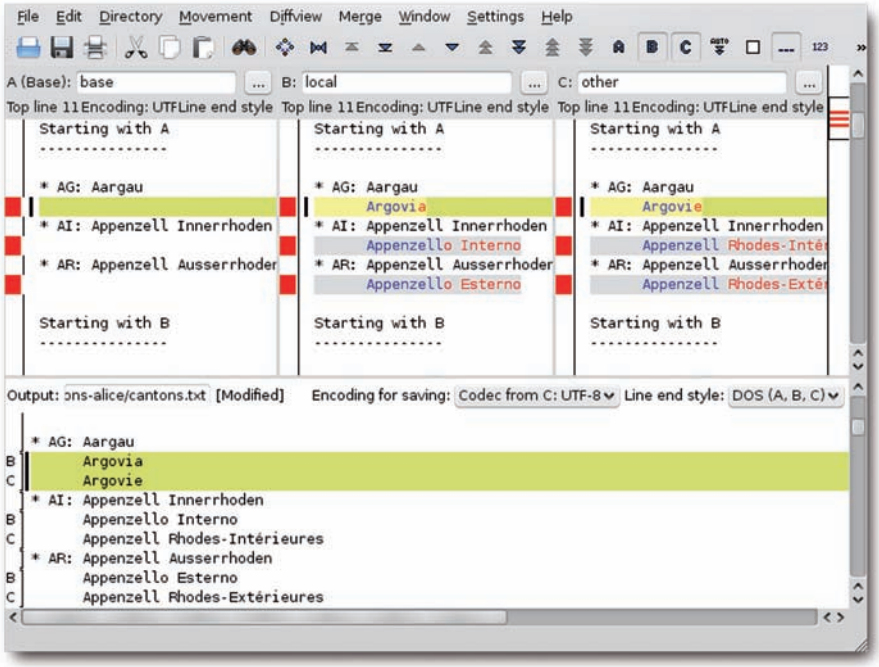

Abb. 3: Merge-Konflikte mit KDiff3 lösen.

Server, auf dem alle Revisionen des Projekts gespeichert sind. Das hat zwei Konsequenzen:

- Die Clients müssen für fast alle Operationen den Server kontaktieren. Subversion etwa kann einzig die Unterschiede zur gerade ausgecheckten Revision anzeigen, ohne zuvor mit dem Server zu sprechen. Einen Server über das Netzwerk anzusprechen, dauert viel länger, als Daten von der lokalen Festplatte zu laden. Änderungen werden daher seltener eingecheckt und die Entwicklung wird behindert. Statt ständig kleine, konsistente Commits zu machen, neigen Entwickler zu umfangreichen Commits, nur um nicht ständig Zeit zu verlieren. Umfangreiche Commits vermischen verschiedene Änderungen, wodurch wiederum die Qualität gefährdet wird.
- Bei einem zentralisierten System ist jede Änderung sofort für jedermann sichtbar. Das führt wiederum zu späten Commits, weil kein Entwickler die Arbeit des restlichen Teams behindern möchte. Hinzu kommt noch, dass man vor jedem Commit einen Merge durchführen muss.

Diese Probleme sollen durch Zweige gelöst werden, wodurch jeder Entwickler seinen eigenen Bereich erhält. Allerdings sind Zweige häufig langsam, die Unterstützung beim Zusammenführen von Zweigen ist schlecht und die Benutzungsschnittstellen führen dazu, dass man den Überblick über die vielen Zweige verliert. Die oft limitierten Benutzerschnittstellen führen dazu, dass man den Überblick verliert. Entwickler, die mit zentralisierten Systemen arbeiten, haben nicht selten regelrecht Angst vor dem Zusammenführen von Zweigen und meiden diese daher von vornherein.

Befehle zum lokalen Arbeiten:
- status: Status der Dateien in der Arbeitskopie anzeigen.
- diff: Unterschiede zur *Working Copy Parent Revision* zeigen.
- commit: Änderungen in der Arbeitskopie als neuen Änderungssatz speichern.
- update: Eine bestimmte Revision in die Arbeitskopie kopieren (häufig nach einem pull).
- log: Historie anzeigen.
- merge: Zwei Entwicklungszweige vereinen.
- init: Ein neues, leeres Repository erzeugen.

Befehle zum Arbeiten mit entfernten Repositorys:
- pull: Neue Änderungssätze von einem anderen Repository ziehen.
- push: Neue Änderungssätze in ein anderes Repository schieben.
- clone: Kopie eines anderen Repositorys erstellen.

Kasten 1: Die wichtigsten Befehle von Mercurial.

Vorteile der verteilten Versionsverwaltung

In Mercurial wird die Historie redundant auf mehreren Computern gespeichert. Jeder Entwickler hat jederzeit lokal vollen Zugriff auf die gesamte Historie. Diese Redundanz – zusammen mit sehr schnellen Operationen – führt dazu, dass die Entwickler ihre Arbeit in vielen kleinen abgeschlossenen Schritten aufzeichnen.

Damit jeder Entwickler unabhängig arbeiten kann, muss er häufig ein *Merge* durchführen. Daher bietet Mercurial einen ausgezeichneten Support, um den Entwickler bei dieser komplizierten Arbeit bestmöglich zu unterstützen. Bereits nach wenigen Tagen wird Merging als natürlicher Teil des Arbeitsflusses angesehen.

Zweige sind ein enorm nützliches Werkzeug und bei verteilten Systemen sind alle Voraussetzungen gegeben, damit die Entwickler sie verwenden. So können sie diese effizient einsetzen, sei es für eine kleine Fehlerbehebung oder um die Hauptlinien der Produktentwicklung zu verfolgen.

Bessere Workflows

Mercurial ermöglicht es dem einzelnen Entwickler und dem ganzen Team, neue Ansätze zu verwirklichen, wie z. B. Workflows über mehrere Teams. Ein Beispiel hierfür ist das Offshoring (siehe Abbildung 4). Bei Subversion haben alle außer einem Team eine langsame Verbindung zum zentralen Server.

Mit Mercurial kann man einen Sammelknoten pro Standort einrichten. Innerhalb eines Standorts können Änderungen so schnell ausgetauscht werden. Die Sammelknoten selbst synchronisieren sich untereinander. Zudem können Regeln festgelegt werden, z. B. für ein Code-Review. Junior-Entwickler schreiben alle Änderungen in ein spezielles Repository, wo sie auf das Review warten (siehe Abbildung 5). Dieses Review kann durch ein *QA*-Team (*Quality Assurance*) oder ein automatisches *Continuous-Integration*-System erfolgen. Nur wenn eine Änderung das Review

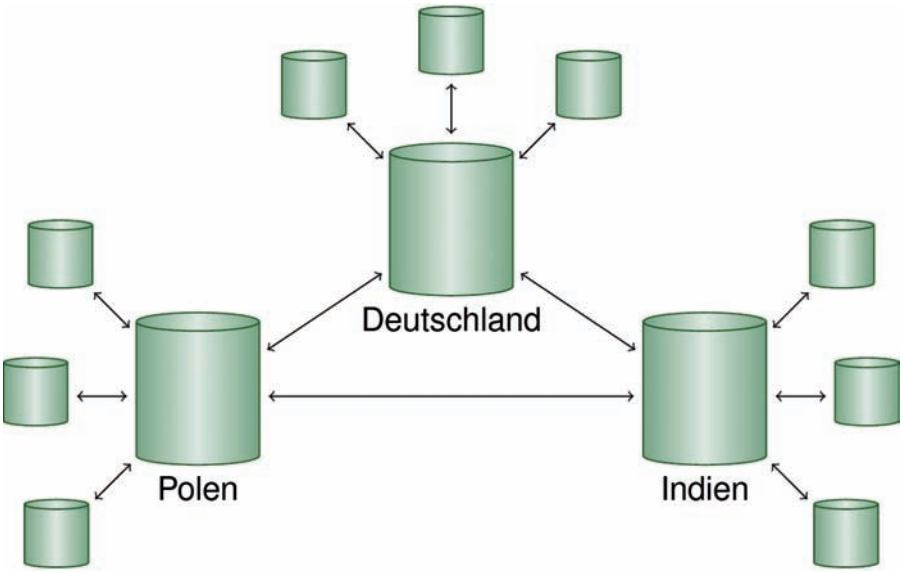

Abb. 4: Global verteilte Entwicklung mit lokalen Sammelknoten.

passiert, wird diese weiter zum Haupt-Repository geleitet – erst dann ist sie für alle Entwickler sichtbar.

Implementierung

Bisher haben wir Mercurial aus der Sicht des Anwenders betrachtet. Sehen wir uns nun einige Details der Implementierung an. Mercurial überwacht zwei Arten von Änderungen:

- Änderungen am Inhalt einer Datei
- Änderungen an der Projektstruktur (hinzugefügte, umbenannte und gelöschte Dateien)

Die Historie wird als Dateiprotokoll (filelog) im Verzeichnis .hg/store/data/ abgelegt. Wenn die Historie eine gewisse Größe überschreitet, wird sie aufgespalten in eine Index-Datei mit dem Namen file.i und eine Datei file.d mit den eigentlichen Daten.

Der Dateibaum wird im Manifestprotokoll (manifest) gesichert. Dieses enthält eine Liste aller Dateien sowie einen Hash-Wert pro Datei. Die Liste ändert sich, wenn Dateien hinzukommen oder entfernt werden. Der Hash-Wert ändert sich, wenn man eine Datei bearbeitet. Um den Dateibaum zu einem bestimmten Zeitpunkt zu bekommen, muss nur die Revision der manifest-Datei rekonstruiert werden, die zum gewünschten Zeitpunkt aktuell war. Jetzt können mit den Hash-Werten in den oben erwähnten Protokollen die Inhalte der Dateien zu diesem Zeitpunkt herausgesucht werden.

Außerdem gibt es noch das Änderungsprotokoll (changelog). Wird ein Commit durchgeführt, speichert Mercurial hier den Benutzernamen, das Datum und die Commit-Message sowie die Revision des Manifest-Protokolls, um die Änderung mit einen bestimmten Zustand im Dateibaum zu verknüpfen (siehe Abbildung 6).

Das Revisionsprotokoll

Der Grundbaustein von Mercurial ist das revlog, kurz für Revisionsprotokoll. Dabei handelt es sich um ein Dateiformat, bei dem Daten immer nur am Ende angehängt werden. Das Revisionsprotokoll wird für Dateiprotokolle, die Manifest-Protokolle sowie das Änderungsprotokoll verwendet.

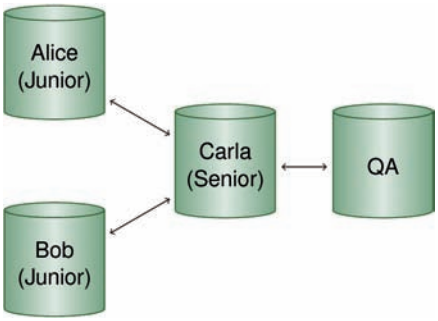

Abb. 5: Beispiel eines Code-Review-Ablaufs mithilfe von „Mercurial Repositorys".

Kommt eine neue Revision hinzu, wird das Delta zur letzten Version berechnet, komprimiert und an die bestehende Datei angehängt. Dieses Vorgehen allein würde allerdings die Rekonstruktion der letzten Revision immer teurer machen. Um das zu verhindern, wendet Mercurial die folgende Strategie an: Wird die Kette der Deltas zu lang, wird statt dem Delta einfach die komplette Datei komprimiert und angehängt. Auf diese Weise kann jede Revision effizient rekonstruiert werden.

Dieses Dateiformat, das sowohl den ursprünglichen Inhalt einer Datei als auch alle Änderungen, die jemals daran vorgenommen wurden, enthält, erlaubt es, Revisionen effizient zu laden, indem direkt zur gewünschten Stelle gesprungen wird und dann alle notwendigen Daten mit einem einzigen Lesebefehl geholt werden. Das Format nutzt so die Eigenschaften moderner Festplatten optimal aus; zudem ist es extrem sicher. Sollte der Computer mitten in einem Commit abstürzen, bleiben die alten Daten unverändert. Der einzige Hinweis auf einen Absturz ist die Tatsache, dass einige Dateien länger als erwartet sind. Dies ist jedoch unproblematisch, da im Änderungsprotokoll keine Referenzen auf diese Daten vorhanden sind (siehe Abbildung 6). Beim nächsten Zugriff auf das Repositiy erkennt Mercurial automatisch, dass die vorherige Operation unterbrochen wurde und schlägt eine Wiederherstellung mittels „hg recover" vor. Das bewirkt, dass die Daten auf die korrekte Länge gekürzt werden. Damit gehören korrupte Repositorys weitestgehend der Vergangenheit an.

Locks vermeiden

Mit der Zahl an Entwicklern steigt die Wahrscheinlichkeit, dass mehrere Personen gleichzeitig Änderungen am zentralen Server vornehmen bzw. dass viele Benutzer die neuesten Daten mit pull holen wollen, während einige wenige gleichzeitig ihre Änderungen dort mittels push ablegen. Mercurial synchronisiert diese Zugriffe möglichst effizient.

Eine Änderung wird erst sichtbar, wenn sich das Änderungsprotokoll ändert. Dadurch können beliebig viele Entwickler gleichzeitig lesen, selbst wenn gerade jemand neue Änderungsprotokolle hoch lädt. Erst wenn der Entwickler, der ein push vornimmt, seine Operation abschließt, wird das Änderungsprotokoll verändert. Ab diesem Zeitpunkt sind die neuen Ände-

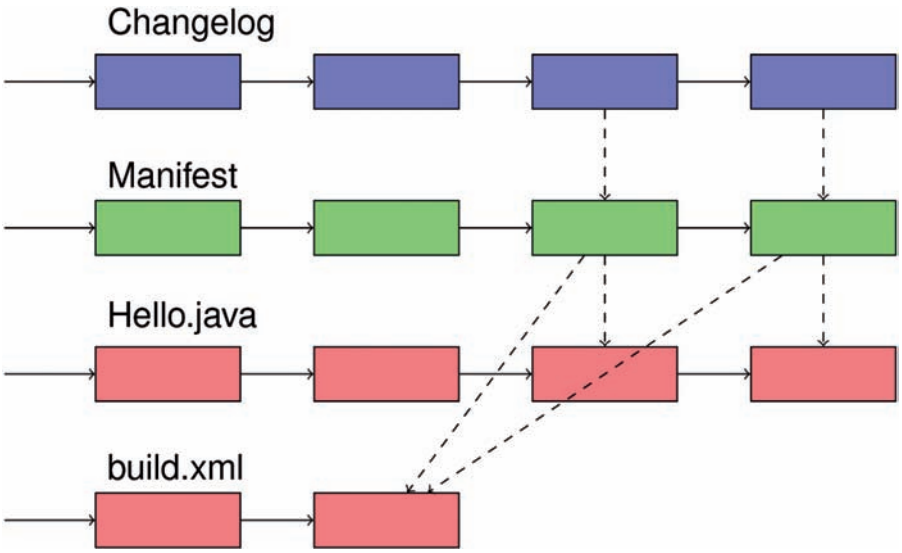

Abb. 6: Abhängigkeiten zwischen dem Änderungsprotokoll, Manifest und den Dateiprotokollen (aus: [O'Sul]).

rungssätze für alle sichtbar; gleichzeitig ist die Struktur des Repositorys wieder vollständig korrekt.

Das bedeutet, dass viele Anwender gleichzeitig lesen können, während nur einer neue Daten anlegen kann. In der Praxis sind Leseoperationen weitaus häufiger als Schreiboperationen – daher sorgt diese Vorgehensweise für optimale Antwortzeiten. Dazu kommt, dass für das Hochladen nur sehr wenig Zeit benötigt wird: Alle aufwändigen Operationen, wie das Komprimieren der Daten und das Berechnen der Deltas, wurden schon vom Client ausgeführt. Daher müssen nur ein paar wenige Kilobyte an Daten übertragen werden, die der Server rasch an die vorhandenen Dateien anhängt – schon ist er wieder bereit für den nächsten Änderungssatz.

Änderungssätze identifizieren

Jeder Änderungssatz bei Mercurial benötigt eine eindeutige ID. Bei einem zentralisierten System beziehen die Clients eine neue ID vom Server. Bei einem verteilten System ist jeder Client isoliert und muss in der Lage sein, eine gültige, weltweit eindeutige ID zu erzeugen.

In modernen kryptographischen Systemen spielen Hash-Funktionen, die auch von Mercurial verwendet werden, eine Schlüsselrolle. Mit der Hash-Funktion kann man einen beliebig langen Input auf eine begrenzte Zahl von Output-Bits reduzieren. Zwei Bedingungen müssen erfüllt sein:

- Aus dem Output darf man nicht auf den Input schließen können.
- Die Wahrscheinlichkeit, dass zwei beliebige aber unterschiedliche Inputs dasselbe Output liefern, soll so klein wie überhaupt nur möglich sein.

Mercurial verwendet den Industriestandard SHA-1 (vgl. [Wik]). Damit kann für jeden Änderungssatz ein Hash-Wert berechnet werden, der diesen eindeutig identifiziert. Dabei werden die folgenden Daten aus dem Änderungssatz verwendet:

- der Hash-Wert aus der manifest-Datei (ergibt sich aus dem Inhalt der Änderung)
- der Name des Benutzers
- das aktuelle Datum
- die Commit-Nachricht
- der Hash-Wert des Parent-Änderungssatzes
- bei einem Merge zusätzlich der Hash-Wert des zweiten Änderungssatzes

Auf diese Weise ergibt sich der Hash-Wert direkt aus den Änderungen, die vorgenommen wurden. Mehrere Entwickler können damit gleichzeitig am gleichen Projekt arbeiten, ohne dass die Gefahr von Überschneidungen besteht. Mehr noch: Da Änderungssätze auch den Hash-Wert von ihren Vorgängern enthalten (dieser Wert enthält wiederum deren Vorgänger), ist es nicht möglich, die Historie des Projekts unbemerkt zu manipulieren, denn sobald jemand auch nur ein Bit ändert, ändern sich sofort alle Hash-Werte aller folgenden Änderungssätze.

Geplante Features

Um Mercurial hat sich eine wachsende Gemeinschaft gebildet, die ständig neue Features entwickelt. Mercurial hat dazu eine genormte Schnittstelle, mit der Erweiterungen einfach eingebunden werden können. Zur Zeit enthält das Basispaket von Mercurial 35 Erweiterungen (vgl. [Cat]). Sie umfassen selten benötigte oder komplexe Features. Damit bleibt der Kern schlank und einfach. Hinzu kommen weitere 80 Erweiterungen, die separat bezogen werden können.

Im Folgenden stellen wir zwei zukünftige Erweiterungen vor, die bald Teil von Mercurial sein werden.

Reduzierte Klone

Von Natur aus wachsen Repositorys und können mit der Zeit ansehnliche Größen erreichen. Der technologische Fortschritt verringert dieses Problem zwar, doch es wäre besser, wenn nur ein Teil der Dateien oder der Historie heruntergeladen werden müsste. Das erste Feature wird *Narrow Clone* genannt, da man nur einen Teil des Dateibaums klont. Das zweite heißt *Shallow Clone*, denn es wird nur ein Teil der Historie geklont.

Mercurial nimmt am „Google Summer of Code", einem von Google organisierten jährlichen Programmierstipendium, teil. Im Sommer 2010 hat ein Student an *Shallow Clones* gearbeitet. Dabei handelt es sich um ein Feature, das häufig von Organisationen gewünscht wird, die sehr alte CVS-Repositorys in Mercurial importiert haben. Dabei wurden oft 10–20 Jahre an Historie angehäuft. Bei der täglichen Arbeit sind diese alten Informationen selten von Bedeutung – daher würde es viel Bandbreite und Plattenplatz sparen, wenn man sie beim ersten Klonen weglassen könnte.

Große Dateien

Es gibt einen Bereich, in dem verteilte Versionsverwaltungen einer zentralisierten Lösung unterlegen sind: bei großen Dateien, wie beispielsweise Videos oder Bildern. Diese sind meist schon komprimiert und können nicht weiter verkleinert werden. Dazu kommt, dass selbst eine kleine Änderung in der Regel zu einer stark veränderten Datei führt, was die Deltas in jeder Revisionen enorm groß werden lässt.

Da die gesamte Historie auf jedem Client gespeichert wird, verschlingt eine große Datei überall Platz und der initiale Klon dauert lange. In einem zentralisierten System kann eine große Datei einfach gelöscht werden, wenn sie nur aus Versehen hinzugefügt wurde. Nachdem die Datei gelöscht wurde, sind Checkouts auf den Clients wieder so schnell wie früher, da sie nur die jeweils letzte Revision vom Server laden müssen.

Eine Lösung ist die Mercurial-Erweiterung „bfiles" (vgl. [Bfi10]). Diese vereint die Stärken verteilter und zentraler Systeme, indem sie große Dateien zentral hält, während kleine Dateien wie gehabt behandelt werden. Wenn man einen Klon zieht, dann bekommt man nur die kleinen Dateien sowie eine Reihe von Proxies. Bei Letzteren handelt es sich um kleine Dateien, die die echten Dateien über ihren SHA-1 Hash-Wert referenzieren.

Fazit

Entwickler sind auch nur Menschen: Wenn man ihnen ein langsames und umständliches Werkzeug gibt, dann ist ihre natürliche Reaktion, dieses so selten wie möglich zu verwenden. Das bedeutet umfangreichere Commits, damit man seltener auf den Server warten muss und den Mitgliedern im Team seltener Probleme bereitet. Verteilte Versionskontrollsysteme lösen das Problem an der Wurzel, indem sie aus jedem Client einen Server machen. Commits sind nun schnell und lokal. Zudem haben die Entwickler die gesamte Historie lokal, was sie unabhängig von Netzwerkprobleme oder Serverausfällen macht und es ihnen erlaubt, effizient nach Fehlern zu suchen. Da Entwickler in der Regel schlau sind, werden sie rasch beginnen sich diese Vorteile zunutze zu machen.

Da Mercurial ein verteiltes Werkzeug ist, muss es sehr gute Unterstützung von Merges bieten. Merges sind einfach und die Commits sind klein – daher kann man öfter ein Merge vornehmen und es gibt weniger Konflikte. Natürlich kann auch Mercurial keine Wunder bewirken, aber es wurde entwickelt, um seine Anwender bestmöglich bei ihrer Arbeit zu unterstützen und nur manuelle Eingriffe zu erfordern, wo es wirklich nicht anders geht.

Um Mercurial herum gibt es eine offene Gemeinschaft. Jeder kann sich in die Mailing-Liste eintragen oder sich im IRC-Channel #mercurial auf irc.freenode.net anmelden, wo man sich online mit den Entwicklern und anderen Benutzern austauschen kann. Die nützlichen Links zu Mercurial haben wir auf der der Seite http://mercurial.ch zusammengefasst.

Dank

Wir möchten uns bei **Greg Ward, Benoit Boissinot** und **Jan Sorensen** für ihre hilfreichen Kommentare zu diesem Artikel bedanken. ■

Links

[Bfi10] Bfiles Extension, siehe: http://mercurial.selenic.com/wiki/BfilesExtension

[Cat] CategoryExtension, siehe: http://mercurial.selenic.com/wiki/UsingExtensions

[Goo10-a] Google, DVCSAnalysis – Analysis of Git and Mercurial, 2010, siehe: http://code.google.com/p/support/wiki/DVCSAnalysis

[Goo10-b] Google, Google Summer Code, 2010, siehe: http://code.google.com/soc/

[Hgl] Hg Init, A Mercurial tutorial, siehe: http://hginit.com/

[O'Sul] B. O'Sullivan, Mercurial: The Definitive Guide, siehe: http://hgbook.red-bean.com/

[Wik] Wikipedia, Secure Hash Algorithm, siehe: http://de.wikipedia.org/wiki/Secure_Hash_Algorithm