Methods

Nog niet zo lang geleden werd er wel eens gesproken over Distributed Version Control Systems, meestal door de collega die normaal stilletjes zijn werk doet. Tijdens de lunch was deze collega plotseling in staat om een betoog te houden over de kwaliteiten van Distributed Version Control Systems, waarschijnlijk met het vuur in zijn ogen en zweetdruppeltjes op zijn bovenlip. Ongetwijfeld glimlachten zijn collega's er schaapachtig bij om daarna weer lekker aan het werk te gaan.

Versiebeheer, maar dan anders

Met Distributed Version Control Systems

Dit artikel is bedoeld voor lezers die nog niet zo bekend zijn met Distributed Version Control. Na het lezen heb je een goed idee wat het is en wat de voordelen van zo'n systeem zijn. Er wordt dan ook geen introductie gegeven met voorbeelden hoe je een en ander in bijvoorbeeld Mercurial of Git doet in vergelijking tot SubVersion. Online en in de boekwinkel zijn bronnen te vinden die veel uitgebreider ingaan op het dagelijks gebruik. Aan het einde van het artikel staan enkele verwijzingen die een goede introductie kunnen bieden op zowel Git als Mercurial.

Java heeft een lange historie met Distributed Version Control. Tot 2010 werd de broncode van de Java SDK opgeslagen in Teamware. Een intern ontwikkeld Distributed Version Control System van Sun. De publieke doorbraak in de Java-community kwam op 1 december 2007. De eerste en (voorlopig) laatste keer dat de username 'duke' iets op de nieuwe Mercurial repository doet, dat kan geen toeval zijn.

```
jdk7-b24 duke [Sat, 01 Dec 2007 00:00:00 +0000]
rev 0 Initial load
```

Een paar dagen later werd Java 7 Build 24 getagged, de eerste release gebouwd vanuit Mercurial. Build 24 was wat betreft features en code exact gelijk aan Build 23 behalve op een punt, het versiebeheersysteem.

2008 was een jaar waarin meer en meer over Distributed Version Control System werd gesproken. Distributed Version Control System was iets voor de echte nerd of leuk voor een opensource project. Maar in deze standpunten is een kentering gekomen. Helemaal doordat opensource projecten meer en meer aan het wisselen zijn naar Distributed Version Control Systems. Tegenwoordig kun je eigenlijk niet meer om dergelijke versiebeheersystemen heen vanwege het feit dat een of meer van de gebruikte frameworks en/of libraries ontwikkeld worden met behulp van een decentraal systeem.

Wijzigingen bijhouden

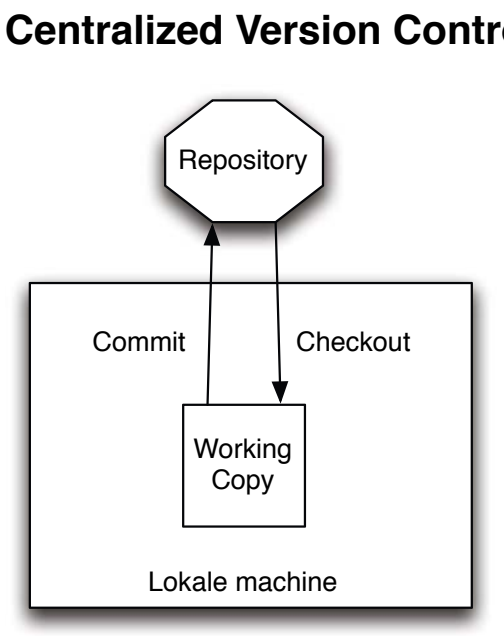
In zijn eenvoudigste vorm is versiebeheer een methode die helpt met het bijhouden van wijzigingen die je maakt op een samenhangende groep bestanden. Denk hierbij aan bijvoorbeeld de broncode van je huidige of laatste project. Als je een kopie van een bestand maakt naar een nieuw bestand met dezelfde naam aangevuld met een timestamp, dan doe je al aan een vorm van versiebeheer. Immers, je houdt alle versies bij en zodoende kun je terug in de tijd gaan en opzoeken wanneer bepaalde wijzigingen zijn geïntroduceerd.

Al dat handwerk is natuurlijk erg foutgevoelig en daar komt een versiebeheersysteem om de hoek kijken. Versiebeheertools houden zich voor ons bezig met het bijhouden van wijzigingen op bestanden. Wij als eindgebruiker moeten nog steeds aangeven wanneer we een afgeronde wijziging klaar hebben. Maar als het systeem een seintje krijgt, dan slaat een versiebeheersysteem de wijziging voor ons op, zodat we later terug kunnen kijken wat we gedaan hebben.

Jeroen Leenarts
is Oracle/Java consultant bij Info Support.

Centralized/distributed

Het belangrijkste verschil tussen een centralized en distributed versiebeheersysteem is de plek waar de gewijzigde bestanden worden opgeslagen en de wijze waarop software-ontwikkelaars de wijzigingen op de bestanden uitwisselen.

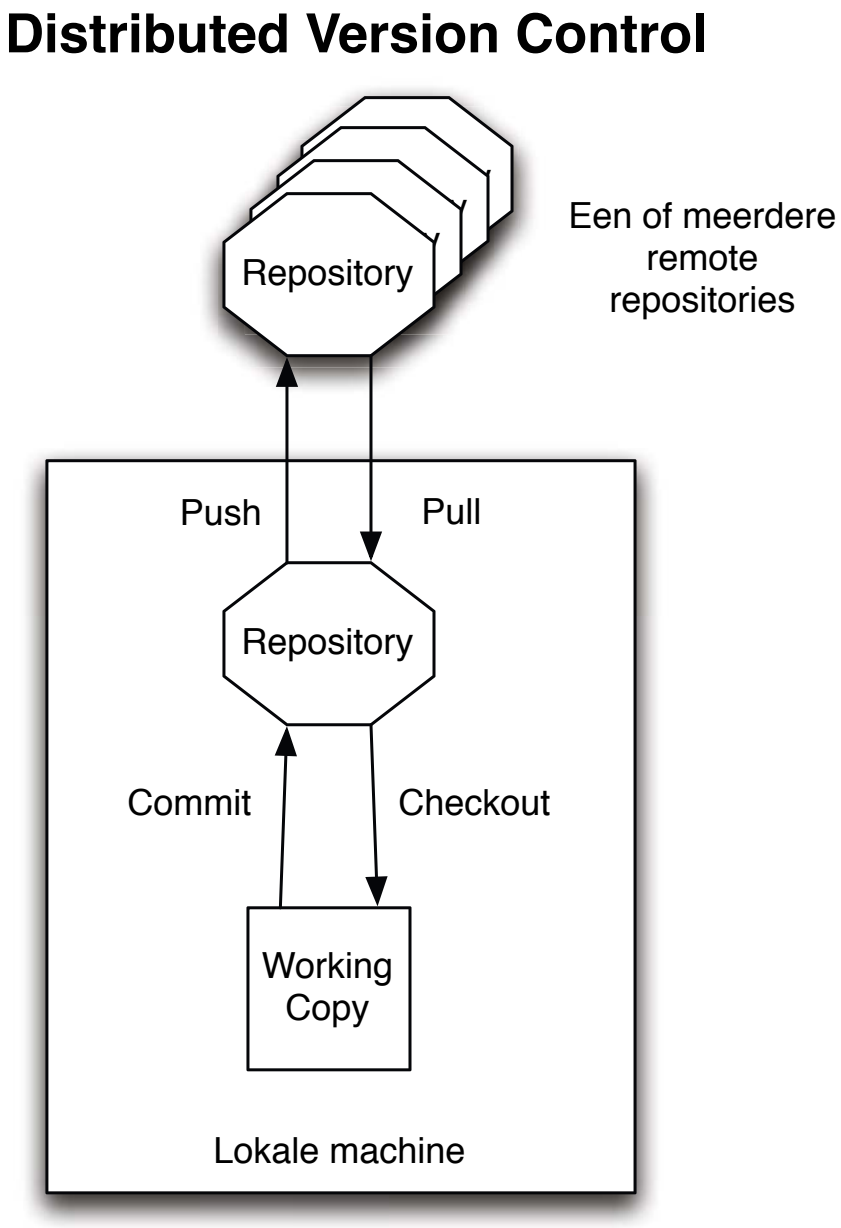


Figuur 1.

Versiebeheersystemen beschikken altijd over een repository; de plek waar wijzigingen en andere metadata worden bijgehouden. Tools als CVS en SubVersion plaatsen hun repository op een centraal punt waar iedere ontwikkelaar zijn wijzigingen tegen synchroniseert. Iedereen maakt verbinding met dezelfde repository, één centrale repository, een gecentraliseerd versiebeheersysteem. Meestal wordt deze centrale repository via het netwerk benaderd. Als de centrale repository niet beschikbaar is, verliest het systeem natuurlijk een hoop functionaliteit.

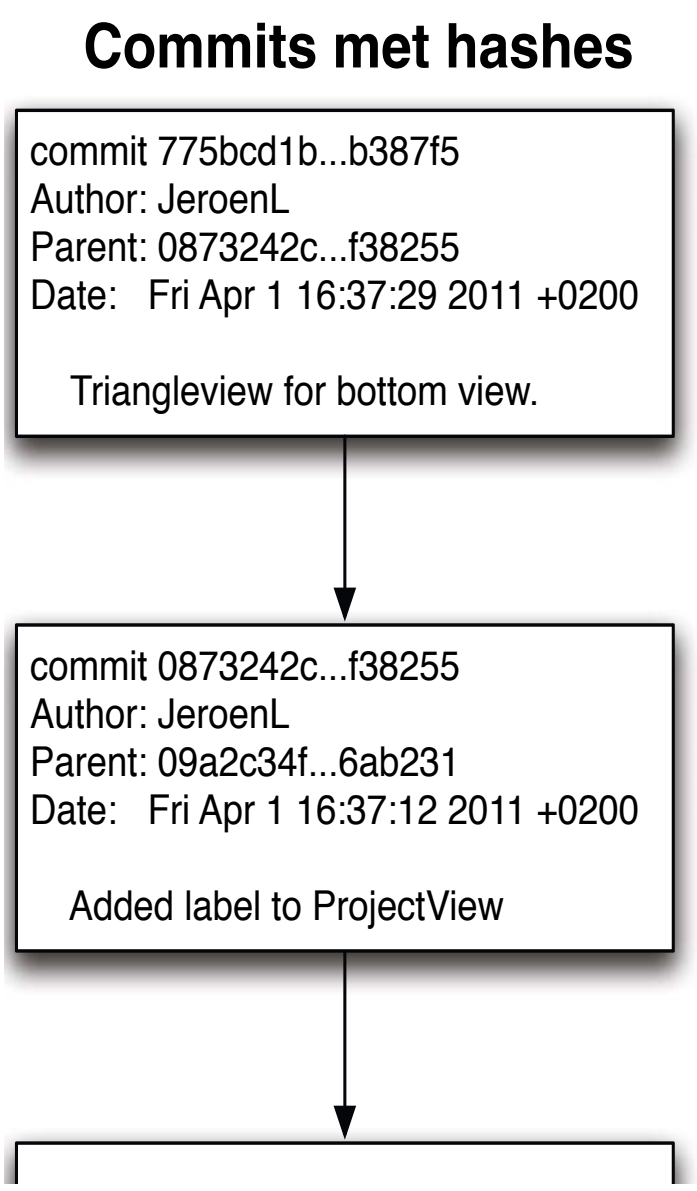
Dit geeft gelijk een probleem aan met gecentraliseerd versiebeheer. Om een wijziging op te kunnen slaan en verder te gaan met je volgende wijziging heb je toegang nodig tot de centrale repository. Je moet dus een succesvolle netwerkverbinding hebben en via dit netwerk toegang hebben tot de centrale repository alvorens je versiebeheerclient het huishoudelijk werk kan doen dat komt kijken bij versiebeheer.

In een Distributed Version Control System heeft iedere ontwikkelaar een complete eigen repository. Sterker nog, op iedere machine waarop hij of zij werkzaamheden voor het project uitvoert kan een unieke repository staan. De wijzigingen die een ontwikkelaar doet, hebben in eerste instantie alleen effect op de eigen lokale repository. Willen andere personen deze wijzigingen benutten, dan zullen deze wijzigingen na het wegschrijven in de repository moeten worden uitgewisseld. Het ophalen van wijzigingen van een andere externe repository naar de eigen repository wordt over het algemeen een pull genoemd. Het plaatsen van wijzigingen vanuit de lokale repository op een externe repository wordt een push genoemd.



Figuur 2.

Er zijn dus veel repositories die allemaal hun eigen wijzigingen krijgen, maar die ook wijzigingen met elkaar uit willen wisselen zonder dat wijzigingen onnodig gedupliceerd worden. Bij Distributed Version Control Systems worden de wijzingen in



Figuur 3.

brokken aangeleverd. Iedere wijziging bevat een cryptografische hash die de wijziging uniek identificeert. En omdat iedere wijziging de hash van zijn ouder bevat, is het mogelijk om deze wijzigingen uit te wisselen en te voorkomen dat er dubbel werk gedaan wordt door het systeem. Sterker nog, het is gegarandeerd te achterhalen of de uiteindelijke broncode daadwerkelijk de juiste is, doordat de oudere hash wordt meegenomen in de berekening van de hash van de wijziging. Hierdoor ontstaat een keten van wijzigingen die volledig kan worden nagelopen. Voor de geïnteresseerden; alle wijzingen vormen aan elkaar gekoppeld middels de hashes een acyclische graaf.

Er kan overigens prima een repository zijn waar iedereen mee synchroniseert, maar dit is iets wat niet afgedwongen wordt door het systeem. De meeste distributed version control systems hebben wel ondersteuning voor een dergelijke opzet met een 'centrale' of master repository.

Voordelen

Ten eerste een kleine hoeveelheid extra complexiteit. Naast checkouts en checkins op je lokale repository moet je nu ook pulls en pushes doen richting andere repositories. In de meeste gevallen is dit echter maar met één enkele master repository.

Voor dat beetje extra werk krijg je wel veel terug:

- Je kunt productief zijn, zelfs wanneer je geen netwerkverbinding hebt. De wijzigingen kun je bewaren in je eigen lokale repository om op een later moment, wanneer je wel een verbinding hebt, alle commits in eenkeer te pushen;
- De meeste operaties zijn een stuk sneller, omdat er geen netwerkverbinding noodzakelijk is, enkel tijdens push- en pullachtige operaties is er noodzaak tot netwerkverkeer;
- Er ontstaat de mogelijkheid tot privaat werken. Vroege versies van nieuwe features hoeven niet gelijk over het hele team uitgestrooid te worden, maar kunnen beperkt blijven tot een persoon of het team waarin deze persoon werkt;
- Automatische back-up, aangezien iedereen met een repository een min of meer up-to-date kopie heeft, zal een verstoring of dataverlies op de centrale server minder gevolgen hebben;
- Ondanks het gedistribueerde karakter en de veelheid aan repositories is het prima mogelijk om een repository aan te wijzen die een speciale status heeft en door iedereen binnen het project als referentiepunt wordt benut. De meeste opensource projecten werken volgens dit model;
- Een verzameling lokale wijzigingen kun je, indien gewenst samenvoegen tot een grotere wijziging. Je kunt dus ongegeneerd iedere mug committen op je lokale repository en deze later tot een wat meer behapbare brok condenseren. Dit wordt squashen (Git) of concatting (Mercurial) genoemd.

Mergen

Door de decentrale aard van Distributed Version Control Systems moet je om de haverklap mergen. Iedereen heeft immers zijn eigen repository en elke repository kan afwijken van ieder ander. Mergen staat van oudsher bekend als een pijnlijke aangelegenheid; complex, tijdrovend en vaak foutgevoelig. Maar omdat Distributed Version Control Systems zijn opgebouwd rondom het verplaatsen van wijzigingen van repository naar repository, zeg maar mergen, gaat mergen in een Distributed Version Control System eigenlijk altijd heel eenvoudig. Alleen de echte conflicten dienen opgelost te worden, alle overige zaken worden probleemloos samengevoegd en bijgehouden en je hoeft je al helemaal geen zorgen te maken over wijzigingen die al eerder via een merge ontvangen zijn.

SubVersion integratie

Als je aan een project werkt dat van SubVersion gebruik maakt dan heb je geluk. Je kunt namelijk prima met Git of Mercurial aan de slag zonder dat de rest van het team het merkt. Beide systemen zijn in staat om een SubVersion repository te benaderen en zich als een volwaardige client te gedragen. Ideaal als je komende zomer op een schaduwrijk plekje van het mooie weer wilt genieten terwijl je aan het werk bent. Al je wijzigingen kun je lokaal committen. Zodra je weer op kantoor bent, kun je een en ander synchroniseren met de SubVersion repository. Ook biedt de SubVersion integratie een elegant migratiepad waarin je niet gelijk de hele buildserver hoeft om te gooien.

Persoonlijk heb ik wel eens bij een complexe merge een complete SubVersion repository gesynchroniseerd met Git, de merge uitgevoerd en vervolgens de resultaten hiervan weer gecommit naar de SubVersion repository.

Keuze maken

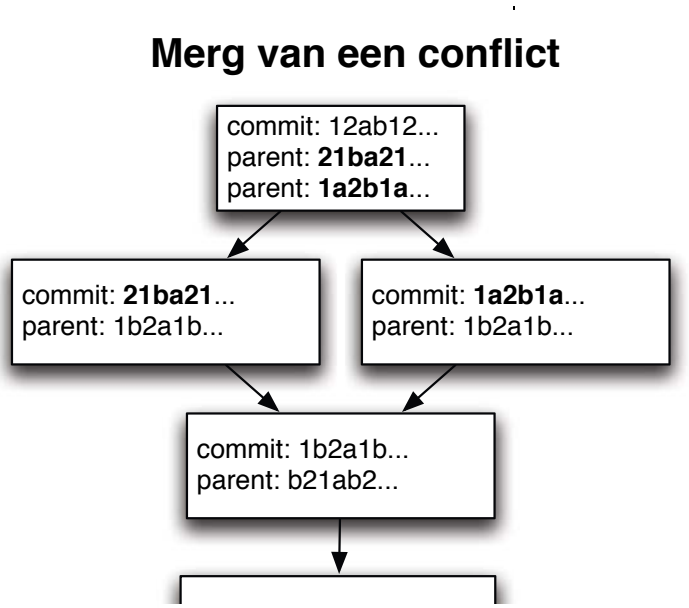
De vraag welk Distributed Version Control System het beste is, is moeilijk te beantwoorden. Wel kan ik je aanraden om op dit moment je keuze te beperken tot Git of Mercurial. Er zijn zeker andere systemen die interessant zijn, maar deze hebben te grote technische beperkingen of een te kleine community om op dit moment levensvatbaar te zijn. Opvallend is dat er geen commerciële distributed version control systems zijn die echt groot zijn. Er zijn er een paar, maar het is maar de vraag of deze een investering waard zijn gezien de opensource en gratis software die beschikbaar is in de vorm van Git en Mercurial.

Mercurial en Git

De twee belangrijkste Distributed Version Control Systems van dit moment zijn Git en Mercurial. Niemand minder dan Linus Torvalds is in 2005 begonnen met de ontwikkeling van Git, omdat het tot dusver benutte BitKeeper niet meer gratis gebruikt kon worden voor de Linux kernel. Git is geschreven in C, shell script en enkele andere tools.

Mercurial heeft dezelfde oorsprong/oorzaak als Git. Mercurial werd enkele dagen na Git aan de wereld voorgesteld. Het oorspronkelijke doel van Mercurial was om Linux kernel ontwikkelaars een alternatief te bieden voor BitKeeper. Het product van de hand van Linus Torvalds heeft dat plan echter onmogelijk gemaakt. Mercurial is geschreven in Python.

Beiden hebben hun voor- en nadelen. In het dagelijks gebruik wordt Mercurial beschouwd als een systeem met een lagere drempel. Je komt een heel eind met bijvoorbeeld de versiebeheerkennis die



Figuur 4.

je hebt opgedaan met SubVersion. Veel commando's van Mercurial komen ook overeen met die van SubVersion. Mercurial wordt ook wel eens beschreven als een soort SubVersion, maar dan ook met Push en Pull.

Git is wat dat betreft een heel stuk complexer. Meer dan 150 commando's die allemaal in je path definitie dienen te staan maken werken met Git wat minder voor de hand liggend.

Wat betreft technische capaciteiten zijn er niet zo heel grote verschillen tussen beide systemen. Sterker nog, intern vertonen beide systemen een werkwijze die conceptueel veel van elkaar weg hebben. Wat wel belangrijk is om te beseffen is dat Git, vanwege zijn opbouw, aanmerkelijk langzamer kan presteren op een Windows omgeving. IDE integratie is goed te noemen voor zowel Git als Mercurial. Voor Eclipse en NetBeans zijn er volwaardige plugins te krijgen voor beide systemen. Deze plugins zijn vaak volledig in Java geschreven. Mercurial zou makkelijker te installeren en te onderhouden moeten zijn, omdat het maar uit enkele executables zou bestaan. Wel dient Python geïnstalleerd te zijn voor Mercurial, maar dit kan via een installatieprogramma eenvoudig met Mercurial mee geïnstalleerd worden.

De keuze tussen Git en Mercurial zal afhangen van de tool support, gebruikte frameworks en persoonlijke of project voorkeur. Zowel Mercurial als Git zijn volwassen en meer dan geschikt voor de taak waarvoor ze geschreven zijn. En als je project SubVersion gebruikt, dan kun je er vandaag nog een eerste experiment mee doen. «

Zowel Git als Mercurial zijn geschikt voor de taak waarvoor ze geschreven zijn.

Links

- http://hginit.com/
- http://hgbook.red-bean.com/index.html
- http://gitref.org/
- http://book.git-scm.com/