RCS, ALEBO AKO TVORIŤ VERZIE PRODUKTU

Je lepšie byť prvotriedna verzia samého seba, ako druhotriedna niekoho iného(July Garland).

Michal Kundrát

Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava
xkundrat[zavináč]gmail[.]com

Abstrakt. *Vo vývoji a údržbe softvérového produktu je dôležitým aspektom správa a udržovanie mnohých verzií a konfigurácií produktu. V mojej eseji sa budem venovať práve systémami na správu verzií, ktorými možno udržovať stav zdrojových kódov, dokumentácie a testovacích dát v rôznych štádiach, čím sa akceleruje a zjednoduší celý proces vývoja softvéru. V eseji sa hlavne venujem otázkam prečo je tak dôležité verzionovať softvérový produkt, akými spôsobmi toto dosiahnuť, a ktoré sú podľa môjho názoru najlepšie systémy pre správu verzií na základe vlastných skúseností. Tiež popíšem môj pohľad na hlavné výhody a nevýhody centralizovaných systémov oproti distribuovaným systémom pre správu verzií.*

Kľúčové slová: *RVC, VCS, CVS, Subversion, Git, systémy pre správu verzií*

Úvod

Neodmysliteľnou súčasťou vývoja softvéru sú zmeny v jednotlivých súboroch projektu. Keďže programátori tvoria, rozširujú a menia jednotlivé časti softvéru, je potrebné tieto zmeny monitorovať. V tomto dokumente sa budem venovať práve systémom, ktoré sa používajú pre správu a kontrolu jednotlivých verzií produktu. Na začiatku bude uvedený hlavný koncept týchto systémov a ich hlavné delenie. Následne budú rozobraté tri najznámejšie a najpoužívanejšie systémy pre správu verzií a moje osobné skúsenosti pri práci s nimi. V závere bude zhrnutá táto tematika a môj celkový pohľad na túto tému.

Systémy pre správu verzií

Systémy pre správu verzií umožňujú monitorovať históriu všetkých súborov projektu. Tieto systémy zjednodušujú a urýchľujú proces tvorby softvéru. Delia sa v zásade na dve hlavné kategórie a to:

- centralizované systémy pre správu verzií
- distribuované systémy pre správu verzií

Každá z týchto kategórií má svoje výhody a nevýhody a pri výbere vhodného nástroja pre správu verzií treba tieto vlastnosti zohľadniť vzhľadom na charakter projektu.

Centralizované systémy pre správu verzií

Názov dostali podľa architektúry, na akej sú postavené. Tieto systémy sú postavené na architektúre klient - server, pričom existuje iba jeden centrálny repozitár (odkiaľ pochádza aj názov pre túto kategóriu), ktorý je vlastne server, na ktorom sú uložené všetky súbory projektu, vrátane ich histórie. Každé štádium je označované ako verzia, alebo revízia. Spôsob, akým je implementovaný stav jednotlivých revízií sa líši v závislosti od konkrétnej aplikácie, avšak spravidla to býva skupina súborov a metadát. Každá revízia je nejakým spôsobom identifikovaná, avšak väčšinou sa používajú celé kladné čísla. Najnovšia, respektíve posledná revízia je označovaná ako *head*/*HEAD*. Klient, ktorý pracuje s repozitárom môže nad ním vykonávať niekoľko základných funkcií, ktoré sa tiež líšia od konkrétnej implementácie.

Jednou z najpoužívanejších funkcií je *checkout*, čo je vlastne akési nahratie stavu repozitára, teda pracovnej kópie do svojho počítača. Používateľ si môže takisto stiahnuť iba niektoré konkrétne súbory, alebo aj celý repozitár. Takisto určiť na základe identifikátora číslo revízie, o ktorú má záujem. Táto pracovná kópia už neobsahuje žiadne informácie o histórii súborov, avšak umožňuje používateľovi ich editáciu. Po editácií súborov má používateľ možnosť odoslať túto novú verziu na server na centrálny repozitár pomocou operácie nazývanej *commit*. Spolu s odoslaním zmien má možnosť pridať aj správu s informáciami, ako napríklad čo zmenil od poslednej verzie, ktoré iné časti programu by mohli tieto zmeny ovplyvniť a podobne. V centrálnom repozitári sa vytvorí nová revízia a číslo revízie sa inkrementuje. Po vykonaní takejto operácie pracovné kópie ostatných používateľov už nie sú v stave, v akom sú v centrálnom repozitári. Aby si zrovnali verzie, musia vykonať operáciu, takzvaný *update*, ktorý vlastne stiahne najnovšiu verziu súborov z centrálneho repozitára.

Zatiaľ by to bolo všetko dosť jednoduché, ak by sme predpokladali, že v jednom čase na produkte robí iba jeden človek. Keďže na produkte robí viacero ľudí, je veľký predpoklad že pracujú paralelne. Toto vlastne aj podnietilo potrebu vývoja takýchto systémov. V minulosti bolo toto riešené metódou *lock-modify-unlock*, čiže súbor na ktorom niekto pracoval sa uzamkol, nikto iný ho nemohol v tom čase editovať okrem dotyčného, a po vykonaní zmien tento súbor opäť odomkol. Už len z princípu jasne vidieť, že síce táto metóda funguje, avšak je veľmi neefektívna a muselo sa nájsť nové efektívnejšie riešenie. Systémy pre správu verzií v prípade, že už aj niekto iný vykonal nad repozitárom zmeny vykonávajú zlúčenie súborov, takzvaný *merge*. Implementácia algoritmu, ktorý sa stará o zlúčenie týchto súborov závisí od konkrétneho systému. Ak sa vyskytnú konflikty, ktoré

nedokáže systém vyriešiť, používateľ je nútený tieto zmeny vykonať vlastnoručne, a až potom je možné vykonať operáciu *commit*.

Hlavný vývoj produktu sa uskutočňuje v hlavnej línii, takzvaný *trunk*. Systémy pre správu verzií umožňujú vetvenie procesu vývoja, ktorý sa nazýva *branche*. Používa sa väčšinou keď používateľ chce do produktu implementovať nejakú novú funkcionalitu, ktorá by mohla program destabilizovať. Ak sa tak aj stane, stane sa to všetko v jednej vetve a tým sa neovplyvnia ostatní používatelia. Niektoré revízie môžu byť dôležitejšie, ako ostatné, a bolo by vhodné ich nejakým spôsobom označiť. Na toto sa používa takzvaný *tag*, ktorý nesie svoj názov a táto revízia sa dá vyvolať pod týmto názvom namiesto čísla revízie. V praxi sa na *trunk*, *branche* a *tag* používajú samostatné adresáre.

Ako si môžeme všimnúť, pri všetkých operáciách je potrebný prístup na internet, respektíve na server, kde sa nachádza repozitár, čo so sebou prináša mnohé nevýhody, ako napríklad to, že sa nedá pracovať off-line. Hlavne pri väčších projektoch môže dochádzať k zahlcovaniu linky, alebo k dlhému času, ktoré potrebujú jednotlivé operácie.

Distribuované systémy pre správu verzií

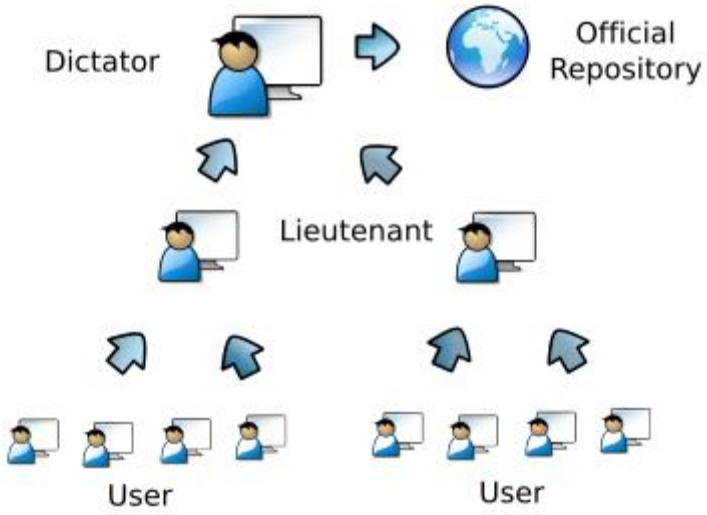
Distribuované systémy pre správu verzií sa líšia od centralizovaných tým, že nepotrebujú centrálny repozitár. Používajú sa lokálne repozitáre, ktoré síce zaberajú viacej miesta ako pracovné kópie v centralizovaných systémoch, avšak prinášajú viaceré výhody a aj s týmto problémom si pomáhajú rôznymi kompresiami dát.

Distribuované systémy využívajú podobné operácie ako centralizované ako napríklad *commit*, *checkout*, *update*, avšak sú neporovnateľne rýchlejšie, keďže sa všetko odohráva na lokálnom repozitári. Ani tieto systémy sa avšak nezaobídu bez prístupu na internet z dôvodu zdieľania. Používateľ môže vytvoriť vzdialený repozitár, ktorý je zdieľaný. Ostatní používatelia môžu nad týmto vzdialeným repozitárom zavolať operáciu clone, ktorá vytvorí na ich počítači lokálny repozitár s rovnakým obsahom, aký má vzdialený. Pre aktualizáciu lokálneho repozitára oproti vzdialenému sa používa operácia *fetch*. Na publikovanie zmien zo svojho lokálneho repozitára na vzdialený sa používa operácia *push*. Na rozdiel od centralizovaných systémov distribuované systémy neobsahujú operáciu *update*, ktorá je dá sa povedať zbytočná, keďže na lokálnom repozitári robí zmeny iba jeden používateľ. Keďže každý lokálny repozitár je nezávislý od ostatných, v distribuovaných systémoch je veľmi dôležité kvalitné zlučovanie súborov.

Na rozdiel od centralizovaných systémov pre správu verzií, ktoré majú všetky veľmi podobný *workflow*, distribuované systémy majú *workflow* omnoho flexibilnejší. Veľmi bežným modelom je jeden zdieľaný repozitár, a pre každého používateľa svoj vlastný lokálny. Tento model sa veľmi podobá na ten, ktorý sa používa v centralizovaných systémoch, avšak prináša so sebou výhody lokálneho repozitára.

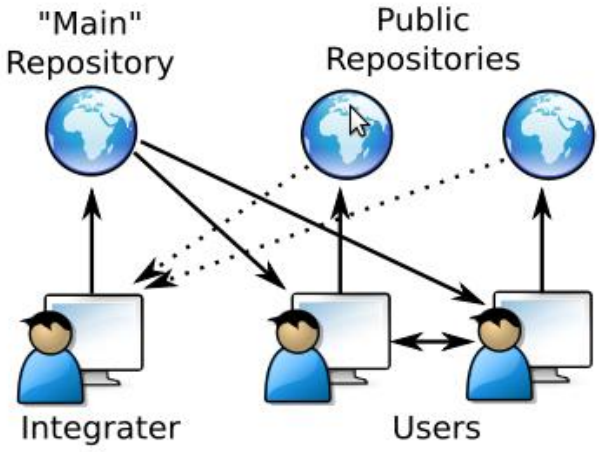
Väčšie projekty, ako napríklad vývoj linuxového jadra, používajú model *lieutenant-dictator*. Model môžeme vidieť na obrázku Obr.1. Každý vývojár sa stará iba o svoj subsystém, pričom každý subsystém má svojho "poručíka". Ten *push-uje* dáta svojmu nadriadenému diktátorovi. Ten sa tým pádom nemusí zaoberať všetkými vývojármi, ale iba svojim "poručíkom". Tento systém funguje na báze princípu dôvery. Každý používateľ, ktorý má na starosti nejakú vetvu dovolí *push-ovať* iba používateľom, ktorým

verí. Diktátor nakoniec zverejní všetky súbory v hlavnom repozitári a všetci si túto novú verziu môžu *pull-nuť* z hlavného repozitára.



Obr.1. Lieutenant-dictator model

Ďalším modelom je *Integration manager* model. Tento model môžeme vidieť na obrázku Obr.2. Každý používateľ má svoj lokálny a svoj verejný repozitár. Pracujú na svojom lokálnom repozitári a na verejný dávajú zmeny, ktoré chcú zverejniť. Následne požiadajú integrátora, aby zapracoval zmeny. Ten môže vybrať *commity*, ktoré on uzná za vhodné, a tieto zmeny zverejní na hlavný verejný repozitár. Z tohto repozitára si všetci používatelia aktualizujú svoje verzie.



Obr.2. Integration manager model

Aplikácie

V tejto sekcii popíšem tri systémy pre správu verzií, dva centralizované a jeden distribuovaný. Konkrétne to bude CVS a SVN ako zástupcovia centralizovaných systémov a Git ako reprezentant distribuovaných systémov. So systémom SVN mám dlhodobejšie skúsenosti. Systém Git používame na našom tímovom projekte, a síce s ním ešte moc skúseností nemám, keďže s ním robím iba krátky čas, pokúsim sa ho priblížiť čo najlepšie. Systém CVS som vybral z dôvodu, že to bol jeden z prvých systémov pre správu verzií a ako prvý zástupca systémov s centrálnym repozitárom.

CVS

CVS ako už bolo vyššie spomínané bol prvým systémom s centralizovaným repozitárom. Vznikol v roku 1984 na univerzite v Amsterdame a autorom bol Dick Grune. Vytvoril systém, aby mohol pracovať so svojimi dvomi študentmi na projekte. V roku 1989 ho prepísal do jazyka C Brian Berlinear. CVS sa postupne stalo najrozšírenejším systémom pre správu verzií, používali ho aj mnohé veľké spoločnosti, a napríklad sourceforge.net ho použil pre viac ako 100 000 projektov.

Aj napriek jeho popularite, CVS stále obsahovalo veľké množstvo obmedzení. CVS napríklad nedokázalo udržovať binárne súbory a vykonávať *diff* nad nimi. Ďalším nedostatkom boli neatomické *commity*. Keďže v tej dobe sa *commit-oval* väčšinou iba jeden súbor, neboli až tak dôležité, avšak s nástupom veľkých projektov obsahujúcich stovky súborov sa pri situáciách, keď napríklad jeden používateľ riešil konflikty v súboroch, a niektoré jeho súbory už boli v repozitári a niektoré nie, a iný používateľ previedol tiež *commit*, repozitár sa mohol dostať do nekonzistentného stavu. Takýto nekonzistentný stav CVS nevedel ani len detekovať. Pri CVS boli veľmi drahé *tag-y* a vetvenie, keďže sa k súborom pridalo veľké množstvo metadát. Tiež nepodporovalo presúvanie a premenovávanie súborov bez straty ich histórie.

Subversion

Subversion, skrátene SVN sa vyvinulo ako nástupca CVS systému, avšak bez jeho nedostatkov, spomenutých v kapitole vyššie. SVN využíva databázu Berkley DB. SVN už podporuje atomické *commity*, teda do repozitára sa prenesú buď všetky súbory, alebo žiadne, čo značne redukuje šancu, že sa repozitár stane nekonzistentným.

SVN už podporuje presúvanie a premenovávanie súborov bez straty histórie, avšak sú tu ešte stále menšie obmedzenia. *Tag-ovanie* a vetvenie už nie je také drahé ako v CVS, pretože SVN používa samostatné adresáre pre *trunk*, *tag* a *branch*, takže sa súbory nemusia obaľovať množstvom metadát. SVN už podporuje aj binárne súbory, ktoré dokáže sám rozpoznávať. Knižnice z ktorých pozostáva SVN by sa dali rozdeliť do troch vrstiev a to:

- klientská vrstva – prostredníctvom nej klient interaguje s repozitárom
- vrstva prístupu k repozitáru – rôzne metódy, akými sa pristupuje k repozitáru
- vrstva repozitára – má na starosti manažment súborového systému na strane serveru

SVN využíva skrytý adresár *.svn*, kde sú uložené napríklad nezmenené súbory pracovnej kópie a takisto informácie o konkrétnej revízii. Na základe týchto informácií dokáže SVN vydedukovať, v akom stave sú súbory. Pri SVN sa súbory môžu nachádzať v štyroch základných stavoch a tými sú:

- Nezmenený a aktuálny – operácie *commit* a *update* neprinesú žiadne zmeny
- Lokálne zmenený a aktuálny – operácia *update* nič nezmení, operácia *commit* prenesie vykonané zmeny do repozitára
- Nezmenený a neaktuálny – operácia *commit* nevykoná žiadne zmeny, operácia *update* stiahne novú revíziu z repozitára
- Zmenený a neaktuálny – *commit* sa nevykoná, a *update* stiahne novú verziu, a pokúsi sa vykonať *merge*, ak sa vyskytnú konflikty, s ktorými si neporadí, je potrebné vykonať zlúčenie manuálne

Git

Git vznikol v roku 2005 pod záštitou Linusa Torvaldsa pre vývoj jadra Linuxu. Jeho hlavnými požiadavkami pre tento systém boli hlavne:

- spoľahlivosť
- výkonnosť
- aby bol distribuovaný
- aby nebol ako CVS

Iba po dvoch týždňoch od začiatku vývoja bol na svete prvý prototyp tohto systému. V roku 2005 bol nasadený na vývoj linuxového jadra. V roku 2007 vyšla verzia 1.5, ktorá priniesla zásadné vylepšenia, ktoré zlepšili dovtedy dosť slabú použiteľnosť. V dnešnej dobe má Git už na starosti Junio C Hamano. Git na rozdiel od väčšiny iných systémov ako napríklad SVN, CVS, Perforce či Mercurial nepoužíva delta kódovanie (rozdiely medzi jednotlivými *commitmy*). Git na udržanie histórie používa "obrazy", ako jednotlivé súbory vyzerali pri jednotlivých *commitoch* v stromovej štruktúre. Toto je veľmi dôležitý koncept na pochopenie, ako funguje Git. Všetky informácie, potrebné pre udržanie histórie sú uložené v súboroch, ktoré sú referencované 40 miestnym číslom. Toto číslo je vlastne SHA1 hashovacia funkcia. Toto prináša so sebou viaceré výhody ako napríklad:

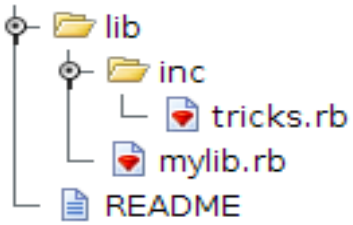
- Git dokáže porovnať, či sú dva objekty rovnaké len na základe porovnania ich mena
- Keďže mená objektov sú počítané v každom repozitári rovnako, rovnaký obsah v dvoch rôznych repozitároch bude mať vždy rovnaké meno
- Git dokáže detekovať chyby len na základe toho, že porovná či názov súboru sedí s SHA1 hash funkciou jeho obsahu

Každý objekt má tri základné polia a to veľkosť, obsah (závisí od typu objektu), a typ. Git rozoznáva štyri základné typy objektov a tými sú:

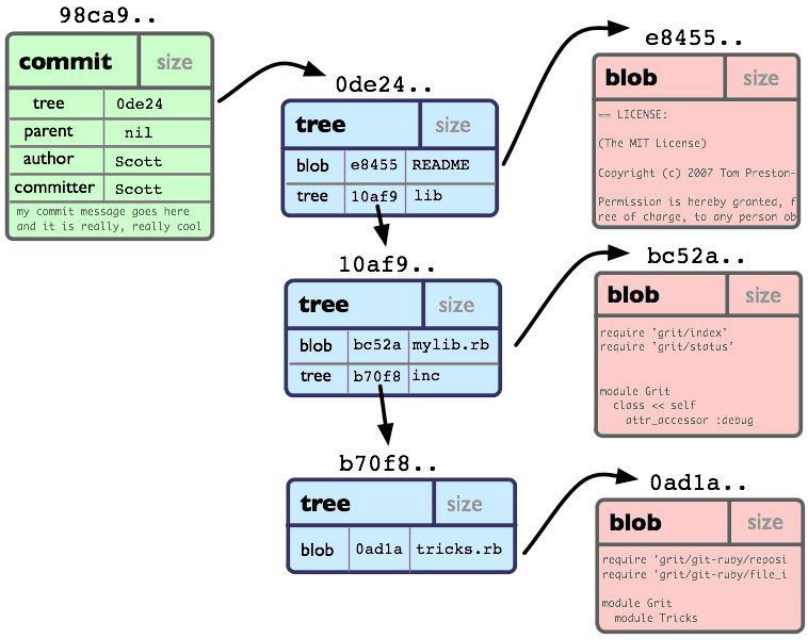
- Blob – využíva sa na uloženie dát súboru

- Strom – je v podstate niečo ako adresár, môže obsahovať iné stromy a/alebo blob-y (podadresáre a súbory)
- Commit – odkazuje na Strom, označujúc tak ako vyzeral projekt v určitom časovom bode. Obsahuje metadáta, ako napríklad časovú známku, autora *commitu* a ukazovateľ na predošlý *commit*
- Tag – spôsob, akým sa označí, nejaký špecifický *commit*

Teda git si vytvára pre udržanie histórie súborov stromovú štruktúru. Ak máme napríklad adresárovú štruktúru ako na obrázku Obr.3, po pridaní do Git repozitáru sa vytvorí štruktúra ako na obrázku Obr.4.



Obr.3. Adresárová štruktúra príkladu



Obr.4. Stromová reprezentácia v Git

Záver

Takže prečo je vlastne dobré verzionovať projekt? Už aj keď nerobíme na projekte v tíme, systémy pre správu verzií nám poskytujú dobrý spôsob, akým môžeme zálohovať náš projekt. Veľkou výhodou oproti klasickým zálohovacím systémom je ten, že môžeme mať mnoho pracovných kópií. Dosť používateľov má viacero počítačov, jeden doma, jeden v práci, prenosný počítač a tak ďalej. Vďaka systémom pre správu verzií tak získavame možnosť používať tie isté dáta od všadiaľ, na hociktorom mieste urobiť zmeny, a tie preniesť opäť na hlavný repozitár. Tým dostávame relatívne jednoduchým spôsobom prenositeľné pracovné dáta s možnosťou sledovania postupných zmien. V projektoch, na ktorých sa zúčastňuje viacej používateľov sa už systémy pre správu verzií stávajú takmer nevyhnutnosťou. Umožňujú kolaboratívne pracovať viacerým používateľom v jednom čase. Zmeny, ktoré vznikajú počas vývoja sa nestratia a sú uchovávané. Vďaka týmto systémom máme neustály prehľad, kto urobil zmenu (vďaka logom z *commitov*), a prečo túto zmenu vykonal (*commit* komentár). Tiež sa môžeme jednoduchým spôsobom vrátiť k hociktorej predošlej revízii.

Ďalšou otázkou je, či použiť centralizovaný, alebo distribuovaný systém pre správu verzií. Pre jednotlivca by som pravdepodobne zvolil SVN, keďže Git obsahuje pre bežného používateľa príliš veľa funkcionality (Git obsahuje viac než 150 príkazov). Pre SVN tiež existuje viacero používateľsky prívetivých aplikácií, ktoré umožňujú s týmto systémom pracovať aj menej skúseným používateľom. S vlastnej skúsenosti poznám napríklad TortoiseSVN, ktorý má naozaj jednoduché rozhranie, vbudované do prostredia windows explorer. Tiež by som SVN zvolil v menších vývojových tímoch, kde majú všetci používatelia neustály prístup k centrálnemu repozitáru. Naopak distribuované systémy sa podľa mňa skôr hodia na veľké projekty, kde sa navzájom používatelia nepoznajú, a model akým majú umožnené pristupovať k jednotlivým repozitárom sa môže upravovať podľa potreby. Takýmito projektmi môžu byť napríklad veľké open-source projekty, ako napríklad už spomínaný vývoj linuxového jadra. Distribuované systémy sa taktiež hodia používateľom, ktorí nemajú možný neustáli prístup na sieť, vďaka lokálnemu repozitáru, ktorý umožňuje pracovať aj off-line. Preto pri výbere systému pre správu verzií treba zohľadniť všetky aspekty podľa povahy projektu a nárokov jednotlivca a na základe toho zvoliť konkrétny systém pre správu verzií, avšak verzionovať projekty na ktorých práve pracujeme odporúčam naozaj každému.

Použitá literatúra

1. Reidar Conradi, Bernhard Westfechtel: SCM: Status and Future Challenges. Norwegian University of Science and technology, Norway
2. Stefan Otte: Version Control Systems. Institue of computer science, Germany
3. Timothy Redmond, Michael Smith, Nick Drummond, Tania Tudorache. Managing Change: An Ontology Version Control System. Stanford University, University of Manchester

Annotation

RCS, or how to create a versions of product

During the development and maintaince of a software product it is important to maintain software revisions, which store state of source codes, documentation, and test data in different project stages. This accelerates the whole process of software developement. In my essay i mainly discuss questions why it is so important to create revisions of system, how can it be possible, and which revision control systems are the best choice in my opinion. I describe the main advateges and disadvanteges of centralized and distributed revision control systems.