Дистрибутирани системи за управление на сорс код

Зорница Атанасова Костадинова, 4 курс, КН, фн: 80227,
Искрен Ивов Чернев, 4 курс, КН, фн: 80246

4 февруари 2011 г.

Цел на проекта

Да запознае читателя с историята и развитието на системите за управление на код (SCM). Ще бъдат сравнени централизираните и дистрибутираните системи както на високо ниво - предимства, недостатъци, сфери на приложение - така и на ниско ниво - обща архитектура, представяне на данните, използвани алгоритми и структури от данни. Ще бъде обърнато специално внимание на дистрибутираните системи за управление на код, тъй като те са по-нови като концепция и вече успешно заместват централизираните системи във все повече проекти, най-забележимо тези с отворен код, но също и в големи компании които по исторически причини използват централизирани системи (google, facebook).

Съдържание

1 Увод **3**

2 История и развитие на SCM **3**
2.1 Ръчно управление 3
2.2 Локално управление 4
2.3 Централизирано управление 4
2.4 Дистрибутирано управление 4

3 Сравнение на архитектурно ниво **5**
3.1 Архитектура на централизираните системи 5
3.2 Архитектура на дистрибутираните системи 8

4 Сравнение на функционално ниво **15**
4.1 Предимства на централизираните системи 15
4.2 Недостатъци на централизираните системи 15
4.3 Предимства на дистрибутираните системи 15
4.4 Недостатъци на дистрибутираните системи 16

5 Заключение **16**

1 Увод

Системите за управление на код (SCM) играят важна роля в процеса на разработване на софтуер. Те съхраняват историята на развитие на файловете като по този начин позволяват на потребителя да прегледа направените промени по различни критерии (времеви период, потребител направил промяната и др.). Правенето на промени по кодовата база също е благоприятствано от факта, че винаги може да се игнорират и състоянието на проекта да бъде върнато към по-старо и стабилно състояние. Историята на развитие на проекта може да бъде използвана и от хора интересуващи се от прогреса по проекта (мениджъри, клиенти), с цел създаване на план за по-нататъшно развитие, оценка за свършена работа и други.

Софтуерните проекти обикновено се развиват в няколко направления:

- едно или няколко стабилни направления - използват се от обикновения потребител;
- направление за тестване (beta версия) - използват се от по-напреднали потребители, които искат да получат нововъведенията колкото се може по-скоро, на цената на по-нестабилно изпълнение;
- направление за развитие - използва се от програмистите докато разработват най-новите промени по кода.

SCM предоставят възможности за управление на отделните направление, като по този начин може да се разграничи кои версии от историческото развитие се използват от програмисти, тестери и обикновени потребители.

2 История и развитие на SCM

Нуждата от SCM се появява през $60^{\text{те}}$ и $70^{\text{те}}$ години на XX век. През това време се появяват първите по-големи софтуерни проекти и става ясно, че поддържането на историята на развитие на проекта е крайно наложително. Развитието преминава през няколко периода:

2.1 Ръчно управление

В началото всеки сам е решавал проблема с пазене на история на проект или отделен файл. Нужен е бил механизъм за запаметяване на състоянието на един или няколко файла в един момент, за да може при нужда да се върне старо състояние. Това може да става със запазване на стара версия на файла с различно разширение (например `filename.old.1`, `filename.old.2` и т.н.) или създаване на архив на цяла директория. Тези подходи вършат работа при малки начинания, но имат големия недостатък че изразходват много памет. Ако един файл има дължина 1 Kb, и е бил запазен 20 пъти по време на своето създаване, тогава заеманата памет е приблизително $1000 * 20/2 = 10\text{Kb}$. Това означава, че заеманата памет нараства линейно с нарастването на запаметените ревизии. На практика обаче, всяка ревизия се различава с малко от предходната, т.е. заеманата памет теоретично може да бъде пропорционална само на размера на проекта, независимо от броя на ревизиите.

2.2 Локално управление

Първите софтуерни продукти за управление на код са работели на ниво отделен файл. Т.е. те са запазвали историята на един файл независимо от останалите. Това може да бъде постигнато като за всяка нова ревизия бъде запазена разликата с предишната. Първата система, която реализира това е SCCS[1]. Тя използва формата припокриващи се разлики (interleaved deltas) за да запазва различията между версиите на един файл. Системата е била разработвана от Bell Labs и се разпространявала чрез заплащане. RCS[2] е наследник на SCCS и набрала голяма популярност през 80$^{\text{те}}$ години, защото била безплатен и развит еквивалент на SCCS. RCS също предоставяла възможност за следене на история на всеки файл по отделно. Освен работната версия на файла се пазел и файл с историята, съдържащ списък с различията еволюирали файла от началното до крайното му състояние.

2.3 Централизирано управление

Системите до момента предоставят възможност за пазене на историята на отделни файлове на компютъра на който са били създадени. Често тази история е трябвало да се споделя между група хора - например екип програмисти. Дистрибутирането на файла с историята между няколко компютъра се оказала не-тривиална задача, особено при наличието на много файлове. Нужна била система, която е в състояние да управлява всички файлове на проекта и да осигури лесен и бърз начин за достъпване на историята на проекта от всеки член на екипа. С тези идеи бил създаден CVS[3] - една от най-известните системи за управление на код, използвана и в момента от немалко проекти. В нея било имплементирано запазването на история за всеки файл, точно както RCS както и централен сървър на който стояла информация за историята на всички файлове. Всеки клиент пазел само една текуща версия на всички файлове и при нужда от други версии се свързвал със сървъра и изтеглял нужната му информация. За да поправи много от проблемите открити с времето в CVS бил създаден Subversion[4]. Subversion има мото "CVS направен както трябва". Тъй като наистина решавал практични проблеми, налични в CVS, а също бил изключително лесен за използване много проекти използващи CVS преминават на Subversion. При Subversion се запазва идеята за централен сървър, на който стои цялата история на всички файлове като при нужда клиентите изтеглят нужната им информация.

2.4 Дистрибутирано управление

Дистрибутираните системи за управление на код приличат на централизирана система, при която всеки клиент пази пълната история на развитие на проекта. Тъй като всеки клиент държи пълно копие на историята и добавя сам нова история трябва да съществува механизъм за обмяна на промените направени между два клиента. Това означава, че може да се симулира модела на работа на централизираните системи като се избере един централен клиент и всички останали синхронизират с него. На практика се използва както централен клиент, така и синхронизиране на два нецентрални клиента, в зависимост от ситуацията. Най-видните примери за дистрибутирани

системи за управление са `Git` [5] и Mercurial[6]. Те набират голяма популярност сред опън сорс обществото, защото позволяват по естествен начин да се споделят промени направени от хора, който не се познават предварително и следователно няма как да споделят общ централен сървър.

3 Сравнение на архитектурно ниво

Модел за едновременен достъп Описва как в случай на едновременна редакция на файлове се избягва въвеждането в хранилището на неверни данни.

- `Схема със заключване` – Промени не са позволени докато потребителят не поиска и получи изключителните права над файла от основното хранилище. Файлът е заключен докато трае текущата редакция.
- `Схема със сливане` – Потребителите са свободни да редактират всички файлове, но при опит да добавят промените си в хранилището са информирани за евентуалните конфликти. В такъв случай системата за контрол на версиите може да разреши конфликта или ако не успее да остави потребителят сам да го разреши.

Разпределените системи почти винаги предполагат модела със сливане.

3.1 Архитектура на централизираните системи

3.1.1 Архитектура на CVS

CVS (съкратено от Concurrent Versions System) е една от първите широко достъпни системи за управление на код с клиент-сървър архитектура. Сървърът пази информация за цялата история на всички файлове от проекта, а клиента може да свали произволна версия от сървъра (check-out), да направи локални промени, след което да качи промените на сървъра (check-in).

Системата е паралелна (concurrent) защото позволява на произволен брой програмисти да работят едновременно—т.е. да правят промени по един и същи проект в едно и също време. Единственото условие е при check-in между последния check-out да няма check-in на друг потребител. Т.е. ако версията свалена от сървъра е най-новата, и не се появи по-нова заради check-in на друг, то качването е успешно. Ако някой друг е създал нова версия междувременно, то останалите трябва първо да свалят новата последна версия, да обединят промените си с нея и да качат след това. В повечето случаи тази операция се извършва автоматично от CVS. Проблем създават само ситуациите при които е редактиран един и същи файл на еднакви места—тогава се налага намесата на програмиста.

Запазване на историята Историята на файловете в CVS repository се запазва за всеки файл по отделно, като се използва RCS формат - т.е. списък с промените довели от началното до крайното състояние. Когато се създава нова версия файловете които са променени увеличават своята версия (т.е промяната им се записва във файла с историята). Файловете участващи във всяка версия не са групирани по никакъв начин. Това е проблем ако искаме

да видим състоянието на проекта във фиксиран момент от миналото—всеки файл пази своята история но информацията за това кои версии на файловете са свързани в ревизиите на целия проект липсва. Това налага ръчното маркиране на ревизии със специални имена, за да може лесно да се възстанови състоянието на всички файлове в определен минал момент.

Връзка със сървъра При качване на промени към сървъра е възможно част от файловете да бъдат обновени, а останалите не—ако например има проблеми с интернета. Това създава проблеми, защото в проект с по-голяма продължителност със сигурност ще се случи, а поправянето става на ръка от администратора.

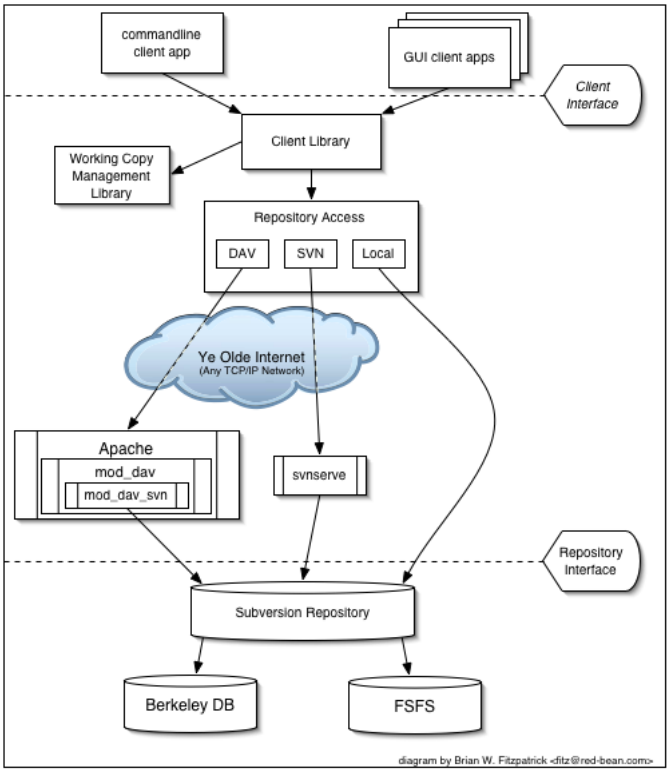
3.1.2 Архитектура на Subversion

Apache Subversion (или само SVN по името на командата svn) е система за контрол на версиите създадена с цел да бъде силно подобна на CVS, поправяйки грешките и имплементирайки някои липсващи нейни функционалности. Използва се широко в средите почитащи отворения код. Сред примерите са Apache Software Foundation, FreeBSD, GCC, Django, Ruby, Tigris.org, PHP, Python и MediaWiki. Google Code и SourceForge предоставят Subversion хостинг за проекти с отворен код.

Функционалности, които го правят предпочитан пред CVS:

- `Атомарни commit-и` - В смисъла на транзакциите. Или цялото множество от промени се регистрира, или абсолютно нищо.
- `Пълна история` на преименовани, копирани или преместени файлове и директории.
- `Метаданни` - Може да се съхранява информация за файлове и директории под формата на двойки ключ - стойност.
- `Двоични файлове` - Съхранението им е ефективно, като от ревизия до ревизия се пази само binary-diff.
- `Branching` е евтина операция независеща от размера на файловете. Механизмът е подобен на хард-линковете в UNIX.
- `Обмен на малко информация` - Протоколът между клиента и сървъра изпраща само diff'ове на файловете в двете посоки.
- `Резултатите` са удобни за парсене. Има възможност за получаване на лог във вид на XML.
- `Конфликти` - За да се избегне необходимостта от разрешаване на конфликти, файловете могат да се заключват. Така програмистът си запазва ексклузивното право само той да ги редактира (reserved checkout).
- `Интеграция` - SVN е написан на C, но поддържа интеграция също със C#, PHP, Python, Perl, Ruby и Java.

Архитектура Моделът е клиент-сървър, дизайнът на библиотеките е слоест.



В долната част на схемата се намира SVN repository-то което съдържа всички данни, които са регистрирани за контрол на версиите. В горната част е представена клиентската програма която управялва локалните копия на тези данни (наречени работни копия). Помежду им са множество маршрути през различни слоеве за достъп до repository. Някои от тях минават през мрежата и през сървъри, които от своя страна достъпват repo-то. Други заобикалят мрежата и директно достъпват repository-то.

Видове хранилища

- `Berkeley DB`
- `FSFS` - Предпочитаният метод за съхраниение на данните, тъй като работи по-бързо и заема по-малко място на диска.

Достъп до хранилищата

- `Местна или мрежова файлова система` - Използва протокола file:///path.
- `WebDAV / Delta-V` - Чрез модула mod_dav_svn за Apache. Протоколът е http(s).
- `Специализиран svn протокол` - Предава обикновен текст (svn://host/path) или криптиран през ssh (svn+ssh://host/path).

Клиенти

- В командния ред
  - Subversion предоставя собствен клиент за командния ред.
- В мениджъра на прозорците
  - TortoiseSVN е разширение на Windows shell което информира за състянието на файловете в revision системата като модифицира иконите в Windows Explorer.
  - SmartSVN работи на подобен принцип.
- Интегрирани в средата за разработване на код
  - AnkhSVN е предвидена за ползване с Microsoft Visual Studio.
  - Subclipse, Subversive работят заедно с Eclipse.
  - Xcode е редактор, включен в Mac OS X 10.5 (Leopard), поддържащ SVN.
- Уеб-базирани
  - Trac - вдъхновена е от CVSTrac и първоначално е наречена "svntrac" заради интеграцията си със Subversion.

Слоеве

Файлова система — Това е най-ниското ниво на което се съхраняват потребителските данни.

Хранилище - Контролира скриптовете които изпълняват действията по контрол на версиите. Тези два слоя заедно съставляват интерфейса на файловата система.

`mod_dav_svn` - Осигурява WebDAV / Delta-V достъп чрез Apache 2.

Достъп до хранилището - Този слой контролира както местния така и отдалечения достъп. На това ниво към хранилищата се обръщаме чрез URL:

- `местен достъп`: file:///path/
- `WebDAV`: http://host/path/; https://host/path/
- `протокол SVN`: svn://host/path/; svn+ssh://host/path/

Клиент, Работно копие на проекта - Най-високото ниво абстрахира достъпа до кранилището и предоставя интерфейс за общи клиентски задачи като автентикация на потребители и сравнение на версии.

3.2 Архитектура на дистрибутираните системи

3.2.1 Архитектура на Mercurial

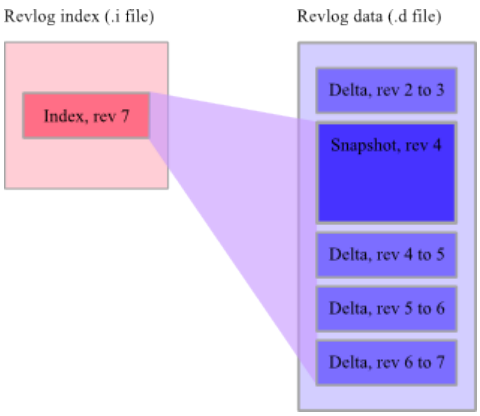
Обща информация Mercurial е многоплатформена, дистрибутирана система за управление на код. Тя е имплементирана главно на Python, но съдържа алгоритъм за двоично запазване на разликите разработен на C. Mercurial работи на всички основни операционни системи: Windows, Unix-like—Linux, OS X, *BSD. При разработването на Mercurial основни цели са били:

- производителност
- скалируемост
- децентрализация
- пълна дистрибутираност
- с адекватно обработване на текстови и двоични файлове
- развити средства за разклоняване и сливане
- концептуално проста

Основни концепции

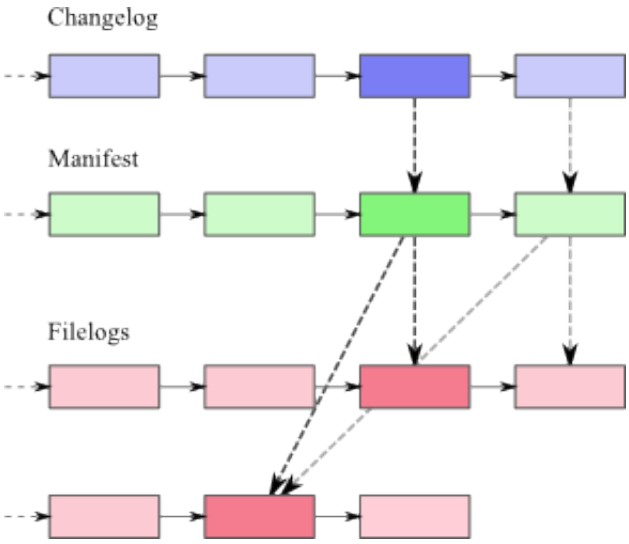
revlog Mercurial пази историята на всеки файл в `revlog` файлове—един индекс, и един за данни. Във файла с данни са записани последователни разлики, или целия файл като се преценява какъв обем данни ще са необходими, за да се възстанови ревизията при нужда—както видео кодерите запазват цял фрейм, последван от няколко разлики с предишния фрейм. Данните се компресират, ако компресирания вариант е по-малък от оригинала. В индекс файла са записани местата във файла с данни в който започва описанието на всяка нова версия (било то разлика с предходната или целия файл), заедно с мета данни.

Всяка ревизия на всеки файл има уникален идентификатор наречен `nodeid`. Той се получава като се сметне SHA1[9] сумата на файла, слепен с `nodeid`-то на предишната версия на файла. Това гарантира, че всеки файл в процеса на своето създаване ще получава винаги различни `nodeid` за различните си версии, даже съдържанието на файла да е напълно еднакво.



manifest Информацията за това кои версии на всеки файл участват във всяка отделна ревизия на проекта се запазва в manifest файл. Една и съща версия на файл може да участва в няколко последователни версии на manifest-а, ако файла не е бил променян в тези ревизии на проекта. Информацията се пази в двоичен формат и представлява `nodeid` на версията на файла заедно с пълното му име. Тъй като manifest-а е файл, който се променя с времето различните му ревизии се пазят в revlog—т.е точно както се пазят промените в нормалните файлове. Това е удачно, тъй като големи части от manifest файла сочат към едни и същи версии на файловете, защото сравнително малко файлове биват променени при всяка нова версия на целия проект. Така последователните ревизии на манифеста се различават с малко, което значително намалява необходимата памет за запазване на всички версии.

changelog Информацията за самите ревзии на проекта се съхраняват в changelog. Информацията включва дата, потребител направил ревизията, предишна ревизия (ревизии в случай на сливане), списък с променените файлове и `nodeid` на manifest файла, съдържащ информация за файловите версии на тази ревизия. changelog файла също се пази в revlog формат.



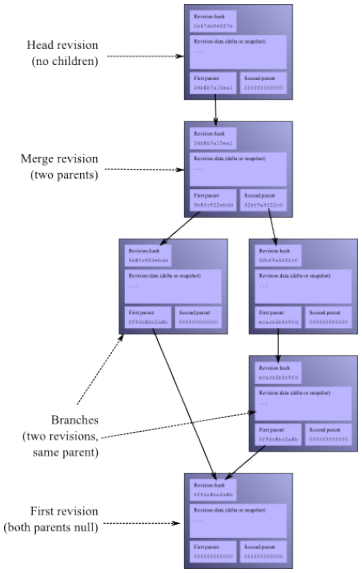
dirstate Състоянието на текущата директория се пази в dirstate. То се използва при записване на промените при създаване на нова ревизия. Пази се кога за последно Mercurial е променял всеки файл и колко е бил размера на файла в този момент. Когато потребителя се интересува от състоянието на работната директория (посредством `hg status`) Mercurial проверява последното време на промяна на всеки файл в проекта и го сравнява с информацията в dirstate. Ако съвпада, тогава се приема, че файла не е бил променян. Ако не съвпада се гледа текущия размер на файла. Ако размера е различен—то файла е бил променен, само в противен случай се налага сравняване на целия файл с последната ревизия запазена от Mercurial.

Начин на работа Когато се наложи да се разгледа състоянието на проекта в определена (вероятно минала) ревизия—например с командата `hg update` се случват следните неща:

1. прочита се индекс файла на changelog и се намира мястото в data файла където стои информация за дадената ревизия на changelog-а
2. възстановява се желаната ревизия от changelog data файла—това включва акумулирането на няколко разлики върху пълна версия—всички взети от data файла
3. прочита се nodeid на manifest-а от changelog за съответната версия
4. възстановява се manifest-а със съответното nodeid—пак чрез използване на index и data файловете му
5. от manifest-а се взима информация за nodeid на всички файлове участвали в проекта за фиксираната версия
6. от index и data файловете за всеки файл посредством nodeid се възстановяват и самите файлове и се записват в текущата работна директория

Разклонения и сливания Основна разлика между дистрибутираните и централизираните системи е лекотата с която дистрибутираните се справят с разклоненията и сливанията в проекта. При централизираните системи всяка ревизия на проект има точно една предишна—което означава, че проекта се развива праволинейно. При дистрибутираните системи е често срещано много хора едновременно да работят върху една и съща начална ревизия (това е разклонение). Това означава, че всички техни промени няма да включват промените на всички останали. Това се случва често и следователно е нужен механизъм по който да бъдат обединени различните промени, за да се получи версия на проекта, в която всички промени са извършени (това е сливане).

В Mercurial всяка ревизия на проекта може да има един или два предшественика. Само един предшественик означава, че върху някаква ревизия са направени промени и така се е образувала нова ревизия (повечето ревизии в един стандартен проект са точно такива). При наличие на ревизии с по един предшественик е възможно дървото на ревизиите да се разклони—ако две промени са базирани на една и съща начална ревизия. За да се слеят две ревизии, които споделят обща бащина ревизия се използва нова ревизия с два предшественика. На базата на графа на ревизиите Mercurial успява да слее повечето файлове без нужда от човешка намеса, защото има възможността да види какви промени са правени по пътя на първия предшественик, и какви промени са правени по пътя на втория предшественик до най-близката обща ревизия. Ако промените са по непресичащи се части от файла може да бъде направено автоматично сливане. Ако промените засягат едни и същи части на един файл—например по едната верига е изтрит даден ред, а в другата е модифициран се налага потребителя да редактира файла на ръка.



Компресия Когато Mercurial запазва файлове на диска се използва deflate[10] алгоритъма, който дава добър баланс между използвано място и време за (де)компресиране. Ако обаче се предават данни по мрежата компресираното съдържание се декомпресира, след което се компресира на ново с друг алгоритъм, който е по-бавен но и по ефективен. Компресира се целия поток от данни което е по-ефективно по време и памет, от компресиране на отделните файлове. Ако за преноса на данни се използва `ssh`[?], то данните въобще не се компресират, защото `ssh` има вградена поддръжка за компресиране. С това Mercurial постига по-добра ефективност върху повече видове мрежи.

Нареждания на входно/изходните операции и атомарност Mercurial гарантира, че клиент четящ от repository-то никога няма да види частични данни, въведени от клиент който пише във същия момент. Това означава, че във всеки един момент може да има един пишещ клиент и много четящи, което значително подобрява времето за достъп. За да се постигне това се използват няколко техники.

файловете се дописват само в края Както вече стана ясно цялата информация в едно repository се съхранява в множество revlog файлове. Записването на нова информация в тях (т.е. добавянето на нова ревизия) е позволено да се извършва само в края на файла. Това подобрява производителността (над методи презаписващи цялото съдържание) и улеснява гарантирането на атомарни операции.

подреждане на входно/изходните операции Когато се записва нова ревизия на проекта реда на записване е следния:

- записват се revlog data файловете на променените файлове от проекта
- записват се revlog index файловете на променените файлове от проекта

- записва се revlog data файла на манифеста
- записва се revlog index файла на манифеста
- записва се revlog data файла на changelog
- записва се revlog index файла на changelog

Както показахме по-горе реда при четене е обратен—това означава, че ако в даден момент клиент четящ repository-то прочита данни сочещи (чрез nodeid) към вече написани данни в следващия файл.

едновременен достъп Подреждането на входно/изходните операции гарантира, че четящ потребител няма нужда да заключва repository-то. Това означава, че четящите потребители имат само нужда от права за четене. Това значително улеснява администрирането на repository-то.

Пишещите клиенти трябва да заключват repository-то. Механизмът за заключване работи на всякакви файлове системи, включително NFS[?], която по принцип създава всевъзможни проблеми за това. Ако пишещ клиент завари заключено repository, той изчаква определено време, проверявайки периодично дали се е отключило. Ако времето изтече и repository-то още е заключено процеса завършва с неуспех. Това е полезно, ако са настроени

3.2.2 Архитектура на `Git`

`Git` е създадена от Линус Торвалдс за целите на разработването на ядрото на операционната система Линукс. При писането ѝ се е обърнало специално внимание на скоростта и лесното включване на код от много участници в проекта. Всяка работна директория в `Git` е пълноправно хранилище с пълна история и възможности за преглеждането ѝ и независима от мрежови достъп до централен сървър.

Дизайн Вдъхновен от BitKeeper и Monotone, `Git` е бил замислен като енджин за система за контрол на версиите на ниско ниво. Над него други щели да надграждат front-end. Проектът обаче се развил дотолкова, че е напълно функционален и за директна употреба. Създателят на `Git` Линус Торвалдс има широки познания върху работата на файловите системи и опит с поддръжката на голям разпределен проект какъвто е операционната система Линукс. Това предопределя следните решения:

- `Разпределена разработка` – `Git` предоставя на всеки разработчик местно копие на цялата база. Промените се включват от едно хранилище в друго като нови разклонения, които могат да бъдат слени по същия начин като местните разклонения.
- `Ефективност при работа с големи проекти` – По думите на създателя си `Git` е бърз и скалируем. Тестове проведени от Mozilla доказват,

че е в пъти по-бърз от повечето системи за контрол на версиите, а достъпът до историята на ревизиите от местното хранилище е 2 пъти по-бърз отколкото от отдалечен сървър. `Git` не става по-бавен с увеличаването на базата от код или акумулираната история на ревизиите.

- `Сериозна поддръжка при не-линейна разработка` – Важно допускане при дизайна на `Git` е, че една промяна по-често ще бъде включвана към проектите на участниците отколкото редактирана. Системата поддържа бързо разклоняване и сливане на код и предоставя инструменти за визуализиране и управление на сложна история на разработка.

- `Съвместимост с протоколи и системи` Медия

  - `ssh`
  - `sockets`

  Протоколи

  - `http`
  - `ftp`
  - `git`
  - `rsync`

  Други системи

  - `cvs` `Git` има възможност за емулиране на CVS сървър и позволява достъпа до хранилища чрез съществуващи cvs клиенти и плъгини за среди за разработка.
  - `svn` Съществуващи subversion хранилища могат да се достъпват с командата git-svn.

- `Криптографска автентикация на историята` – Името на всеки commit зависи от пълната история довела до него. Веднъж направен commit-а не е възможно промяна да не се отрази на криптографския хеш, а следователно и на името на ревизията. В този смисъл промените по `Git` са immutable.

- `Модулен дизайн` – `Git` е създаден като множество програми на C, свързани с shell скриптове - wrapper-и около тези програми. Макар и сега повечето от скриптовете да са пренаписани на C като част от усилието да се направи `Git` напълно функционален под Windows, дизайнът е останал, което прави лесно навръзването и замяната на отделни компоненти.

- `Стратегиите за сливане са лесно заменими` – Като част от модулния дизайн, `Git` ясно дефинира незавършено сливане и има множество алгоритми за справяне с него. В случай на невъзможност да се слеят програматично промените е необходима ръчна редакция от страна на потребителя.

- `BitKeeper като модел` – Използваната в първите години на разработване на Линукс ядрото система е модел за интерфейса и потока на действия в `Git` . Технически двете са много различни, което е била една от целите които създателят на `Git` си е поставил за да бъде очевидно, че макар и създаден да го замести, `Git` не е копие на BitKeeper.
- `CVS като анти-модел` – Централизираната система се оказва неприложима за целите на Линус.
- `Сигурност` – Гаранции срещу повреда на данните, била тя случайна или злонамерена.

4 Сравнение на функционално ниво

4.1 Предимства на централизираните системи

4.2 Недостатъци на централизираните системи

4.3 Предимства на дистрибутираните системи

- Позволяват ефективна работа дори когато потребителите не са свързани в мрежа.
- Включването в проект е лесно - не изисква позволение за писане от привилегировани потребители.
- Използването за лични проекти е лесно и удобно. Подходящо за началните фази на проект, когато потребителите все още нямат нищо готово за публикуване.
- Не се създава единична централизирана контролираща система, която да създаде проблем в случай на срив. Всяко работно копие на базата може да служи за отдалечен backup на базата и историята на промените ѝ, намаляващо риска от загуба на данни.
- Все пак се позволява централизиран контрол когато е необходимо да се издаде официална версия на проекта.
- Повечето операции са много по-бързи отколкото в сентрализираните системи, тъй като не използват мрежата.

Отворени системи Дистрибутираните VCS са подходящи за използване от отворени системи заради:

- `независимите разклонения`
- `лесното сливане на отдалечените хранилища`

При отворените системи:

- Всяко разклонение практически се имплементира като работно копие. Сливанията се извършват чрез размяна на patch-ове между отделните разклонения.

- Програмистите имат по-голяма готовност да създават нови версии на проекта когато е необходимо. Всъщност работното копие на хранилището само по себе си е потенциална нова версия. Вслучай, че неразбирателствата се изгладят сливането на кода в едно е лесно.
- Възможно е да се вземат само отделни промени (cherry-picking) селектирани от различни потребители.
- Не е задължително да съществува оригинално, отправно и единствено вярно копие на базата с код. Вместо това съществуват множество работни копия. В този смисъл е лесно да се създадат и множество "централни" хранилища.
- Код от отделните хранилища се слива на базата на т.нар. "мрежа на доверието" (web of trust). Исторически е доказано, че това подобрява качеството на кода.

4.4 Недостатъци на дистрибутираните системи

- Първоначалното клониране на хранилището на локалната машина е по-бавно в сравнение с централизираните системи, тъй като се копират всички разклонения с цялата им история.
- Липсват механизми за заключване, каквито традиционно са част от централизираните системи. Това е важно при двоични файлове, които не подлежат на сливане, например изображения.

5 Заключение

Литература

[1] http://sccs.berlios.de/

[2] http://www.gnu.org/software/rcs/

[3] http://savannah.nongnu.org/projects/cvs

[4] http://subversion.apache.org/

[5] http://git-scm.com/

[6] http://mercurial.selenic.com/

[7] http://mercurial.selenic.com/wiki/Design

[8] http://mercurial.selenic.com/wiki/Design

[9] http://en.wikipedia.org/wiki/SHA-1

[10] http://en.wikipedia.org/wiki/DEFLATE

[11] http://ftp.newartisans.com/pub/git.from.bottom.up.pdf