Ari & Tero Roponen

Avoimen lähdekoodin versionhallintaohjelmat

Tietotekniikan
(ohjelmistotekniikka)
pro gradu -tutkielma
26. marraskuuta 2007

JYVÄSKYLÄN YLIOPISTO
TIETOTEKNIIKAN LAITOS

Jyväskylä

Tekijä: Ari & Tero Roponen
Yhteystiedot: {ari,tero}.roponen@gmail.com
Työn nimi: Avoimen lähdekoodin versionhallintaohjelmat
Title in English: Open Source Version Control Systems
Työ: Tietotekniikan (ohjelmistotekniikka) pro gradu -tutkielma
Sivumäärä: 238
Tiivistelmä: Versionhallintaa on tutkittu jo yli 30 vuotta ja sitä pidetään yhtenä kehittyneimmistä tietotekniikan osa-alueista. Viimeisten vuosien aikana mielenkiinto versionhallintaa kohtaan on kuitenkin vähentynyt, ja tutkimus on keskittynyt konfiguraationhallintaan. Samanaikaisesti avoimen lähdekoodin piirissä on syntynyt uuden sukupolven versionhallintaohjelmia, jotka sopivat erinomaisesti hajautettuun ohjelmistonkehitykseen. Tässä tutkielmassa tutustumme versionhallinnan perusteisiin ja siitä tehtyyn tutkimukseen. Katsomme, miten versionhallinta on kehittynyt avoimen lähdekoodin yhteydessä, ja miten vastaantulleita ongelmia on ratkaistu. Tavoitteenamme on selvittää, onko avoimen lähdekoodin versionhallintaohjelmien kehittäminen tuonut versionhallinnan alalle uusia aiheita tutkittavaksi, vai onko ala todella niin hyvin tunnettu kuin usein väitetään.

English abstract: Version control has been under research over 30 years. It is considered as one of the most mature field of computing. In the last few years, interest in version control research has diminished as research has concentrated on configuration management. Meanwhile, inside the circle of open source, there has evolved a new generation of version control software that is excellently suited for distributed software development. In this thesis, we review the basics of version control and the research about it. We look at some open source version control tools and find out how they solve some practical problems. Our main goal is to determine if the work in open source version control software has brought new ideas available for research, and if the version control really is as well understood as is often claimed.
Avainsanat: avoin lähdekoodi, versionhallinta
Keywords: open source, version control

Esipuhe

Kiinnostuksemme versionhallintaan heräsi ns. BitKeeper-kriisin johdosta vuonna 2005, jolloin Linux-ytimen kehitys jäi versionhallinnan osalta tyhjän päälle. Muiden innokkaiden tavoin myös me löysimme itsemme koodaamassa omaa järjestelmää, DataKeeperiä, jonka kuitenkin kaikkien onneksi pian kuoppasimme, kun Linus Torvalds ilmoitti omista suunnitelmistaan. Siitä hetkestä lähtien olemme seuranneet versionhallinnan kehittymistä ja päässeet iloitsemaan siitä, että tälläkin alueella tapahtuu taas edistystä.

Tämän opinnäytetyön tarkoituksena ei ole ollut täyttää ainoastaan tutkinnollisia tavoitteita vaan myös omissa tiedoissamme olleita aukkoja. Monissa tapauksissa muiden auttamiseen ei nimittäin riitä pelkkä innostus, vaan siihen vaaditaan myös syvällistä ymmärtämistä – niin asioiden kuin ihmistenkin.

Onneksi tällaisia ymmärtäviä ihmisiä vielä löytyy: tämän työn ohjaajat Timo Männikkö ja Timo Tuunanen ovat olleet kärsivällisiä, ja vanhemmiltamme on löytynyt ymmärtämystä siihen, että ammatillisen tutkinnon jälkeen jatkoimme vielä teoreettisten opintojen parissa.

> "Yksi näkee paperiteollisuuden raaka-aineeksi sopivia betula-suvun kasveja, toinen huonekaluihin sopivia acer-suvun edustajia, kolmas parketeiksi kelpaavia quercus-suvun kasveja ja neljäs luonnon monimuotoisuudelle tärkeitä populus-suvun yksilöitä. Näitä kaikkia ymmärtävä näkee metsän puilta – kirjaimellisesti."

Niemisjärvellä 26. marraskuuta 2007,

Ari Roponen *Tero Roponen*

Sisältö

Kuvat

Taulukot

1 Johdanto

> *While computer science is perhaps too young to have brought forth grand theories, my greatest fear is that the lack of such theories might be caused by a lack of experimentation. If scientists neglect experiment and observation, they'll have difficulties discovering new and interesting phenomena worthy of better theories.*
>
> — *Walter F. Tichy*

Ohjelmiston konfiguraationhallintaa (SCM, Source Code Management / Software Configuration Management [1]) on tutkittu jo usean vuosikymmenen ajan. Aiheesta on kirjoitettu lukuisia tieteellisiä artikkeleita ja siitä on pidetty useita konferensseja ympäri maailman[2]. Itse asiassa aiheesta on olemassa jo niin paljon tutkimusta, että joidenkin tutkijoiden mielestä ainakin tietyiltä osa-alueilta on jo saatu kaikki oleellinen selville. Yhtenä tällaisena hyvin ymmärrettynä alueena pidetään *komponenttitason konfiguraationhallintaa* eli tutummin *versionhallintaa* (VC, version control). [19]

1.1 Versionhallinnan nykytila

Tietotekniikan alkuaikoina versionhallintaa hoidettiin ad hoc -tyylisesti, eli uusia menetelmiä ja käytänteitä kehitettiin tarpeen mukaan. 1970-luvun alkupuolella tilanne kuitenkin muuttui, kun käyttöön saatiin ensimmäinen varsinainen versionhallintaohjelma, Marc J. Rochkindin kehittämä *SCCS (Source Code Control System)*. Nyt yli 30 vuotta myöhemmin käytössä on kymmeniä erilaisia järjestelmiä, joista suurimmassa osassa käytetään edelleenkin 1970-luvulla esiteltyjä ideoita. [15, 56, 57]

1990-luvun lopulle tultaessa versionhallinnan teoreettinen perusta oli osoittautunut niin vankaksi, ettei siihen liittyviä mullistuksia ollut enää odotettavissa. Asia

[1] Nämä näyttäisivät olevan tämän lyhenteen yleisimmät tulkinnat. Se, mitä SCM:llä sitten oikeastaan tarkoitetaan, riippuu siitä, keneltä asiaa kysytään. Tulkinnasta riippuen esimerkiksi sekä RCS että Vesta voidaan luokitella konfiguraationhallintaohjelmiksi, vaikka niiden erot ovatkin melkoiset: RCS käsittelee vain yksittäisiä tiedostoja, mutta Vesta ottaa huomioon myös mm. kääntäjien versionumerot sekä ympäristömuuttujien sisällöt.

[2] ks. http://www.cs.ucsc.edu/~ejw/scm12/scm12history.php (viitattu 26.7.2007).

voidaan kiteyttää Frühaufin ja Zellerin sanoin vuodelta 1999 näin [19]:

> SCM is a mature discipline. It is mature in practice, as it is successfully used. And it is mature in research, since there is much to be taught–and not so much left to be researched.

Tämän jälkeen voisi aivan hyvin ajatella, ettei versionhallinnalla ole enää mitään annettavaa tämän vuosituhannen tieteelle. Aivan näin yksimielisiä asiasta ei kuitenkaan olla, sillä muutamat ennakkoluulottomat tutkijat ovat nostaneet versionhallinnasta esille erään osa-alueen, jota aiemmin on suorastaan kartettu: käytännön sovelluksiin tutustumisen sekä kokeellisen ongelmanratkaisun.

Artikkelissaan "Should Computer Scientists Experiment More?" [63] Walter F. Tichy esittää käsityksensä, jonka mukaan myös tietotekniikan tutkimuksessa pitäisi tehdä kokeiluja ja havaintoja, samaan tapaan kuin perinteisissä kokeellisissa luonnontieteissä. Hänen mukaansa tietotekniikka ei keinotekoisuudestaan huolimatta ole eksaktia, joten kaikkia siihen liittyviä asioita ei voi ymmärtää pelkkien teorioiden avulla. Frühauf ja Zeller ovat samalla linjalla, ja jatkavatkin edellä esitettyä lausumaansa näin [19]:

> [...] it was amazing to see how few hard facts were available to back specific judgements. The most important result of our assessment is that many more SCM experience reports and experiments are needed–*we need to know what we know before we can move on*.

Asia voidaan ymmärtää niin, että tutkijat ovat nykyään kiinnostuneita ns. *hiljaisesta tiedosta*[3], joka on jäänyt tutkimuksen ulkopuolelle, mutta jota käytetään menestyksekkäästi joka päivä. Tämän tietämyksen saattaminen tutkimuksen piiriin voisi hyvinkin antaa Tichyn kaipaaman sysäyksen uusien teorioiden kehittelyyn.

1.2 Hiljainen tieto

Perinteisessä teollisuudessa tutkimus ja tuotekehitys pidetään tiukasti erillään varsinaisesta tuotannosta. Uusia ideoita ei oteta mukaan enää tuotantovaiheessa, sillä niiden pitää ensin läpäistä vaaditut tutkimus- ja kehitysprosessit, ja olla muutenkin

[3] Yksinkertaisuuden vuoksi hiljaisella tiedolla tarkoitetaan tässä yhteydessä mitä tahansa sellaista tietoa, jonka olemassaoloon ei yleensä kiinnitetä huomiota. Luvussa 4 tätä asiaa käsitellään kuitenkin tarkemmin, joten käsitteen todellinen merkitys selviää myöhemmin.

määräysten mukaisia. Kaiken tuotantoon asti päässeen pitää olla tutkitusti toimivaa ja mahdollisimman yllätyksetöntä, jotta lopputulos olisi suunnitelmien mukainen. Mahdollisia tuotannon aikana esiintulevia korjaus- ja parannusideoita sovelletaan vasta tuotteen seuraavissa versioissa, mutta osa ideoista jää ns. hiljaiseksi tiedoksi, jota ei kirjata minnekään.

Ohjelmistoteollisuudessa käsitellään perinteisestä teollisuudesta poiketen aineettomia asioita, joten näiden väliltä löytyy selviä eroja. Aineellisista tuotoksista poiketen tietokoneohjelmia voidaan muokata varsin vapaasti, joten lähes kaikkea voidaan säätää vielä aivan loppumetreillä. Muutosten teon helppoudesta seuraa, ettei kaikkea yleensä ajatella loppuun saakka, jolloin tehtyjä virheitä voidaan joutua korjaamaan tuotteen julkistuksen jälkeenkin. Virheiden välttämiseksi tällaista "liian vapaata" muokkaamista on alettu rajoittaa erilaisten prosessimallien avulla, joiden ansiosta ohjelmistokehityksestä on tullut nykyinen vakavasti otettava, kurinalainen teollisuudenala.

Avoimen lähdekoodin puolella tilanne on toisenlainen. Kuten myöhemmin osoitetaan, tämän liikkeen piirissä ohjelmointia pidetään yhtenä itseilmaisun muotona, jota ei pidä yrittää keinotekoisesti rajoittaa joihinkin muotoihin. Tämän vuoksi eri prosessimalleista kerättyjä ideoita sovelletaan vain niiltä osin, kuin niiden katsotaan edesauttavan halutun lopputuloksen syntymistä. Tästä seuraa se, että avoimen lähdekoodin projekteissa tehdään paljon sellaisia kokeiluita, joihin perinteisessä ohjelmistoteollisuudessa ei ikinä uskallettaisi ryhtyä. Näistä rajoja rikkovista kokeiluista on kuitenkin arvaamatonta hyötyä, ja niitä voidaan pitää teknologisen edistymisen kannalta jopa välttämättöminä [58]:

> The freedom to conduct experiments is essential to any society that has a serious commitment to technological innovation or to improved productive efficiency. The starting point is that there are many things that cannot be known in advance or deduced from some set of first principles. Only the opportunity to try out alternatives, with respect both to technology and to form and size of organization, can produce socially useful answers to a bewildering array of questions that are continually occurring in industrial (and in industrializing) societies.

Tämä teknologisen kehityksen ja talouden suhteita tutkineen professori Nathan Rosenbergin lausuma on aivan samoilla linjoilla aiemmin mainittujen tutkijoiden kanssa siinä, että edistymisen – sekä tieteen, teknologian että talouden – kannalta riskialttiit kokeilut ovat usein välttämättömiä. Monissa yrityksissä niitä toki tehdään-

kin, mutta niiden tulokset eivät useinkaan päädy julkisuuteen muuten kuin valmiin tuotteen muodossa, ja silloinkin patenttien suojaamana. Vapaasti hyödynnettävissä olevia ideoita löytyykin usein vain yksilötasolla tai harrastusmielessä tehdyistä projekteista, mutta yksi ilahduttava poikkeus sentään löytyy: avoimen lähdekoodin ohjelmistoprojektit.

1.3 Tutkimusongelma

Avoimen lähdekoodin projektien luonteen ja edellä mainittujen seikkojen perusteella voidaan päätyä siihen, että näillä projekteilla on hallussaan sellaista tietoa, joka olisi tieteellisesti merkittävää. Tätä hiljaista tietoa kyllä hyödynnetään projektien sisällä, mutta se palvelee vain käytännön päämääriä; kokeilemalla hankittua tietämystä ei pidetä niin merkittävänä, että sitä kannattaisi esitellä ulkopuolisille.

Oezbek ja Prechelt [48] huomauttavat, etteivät avoimen lähdekoodin parissa liikkuvat kehittäjät ole kiinnostuneet tieteellisistä tutkimustuloksista vaan ensisijaisesti ohjelmien kehittämisestä. Tästä voi tehdä sen johtopäätöksen, etteivät nämä kehittäjät ole myöskään innokkaita laatimaan julkaisuja omista löydöistään, vaikka pitäisivätkin niitä merkittävinä. Tutkijat eivät puolestaan ole kiinnostuneita yksittäisten sovellusten toiminnoista, sillä he haluavat käsitellä asioita yleisemmällä tasolla.

Joskus voi käydä niin, että jonkin käytännön ongelman ratkaisu johtaa lopputulokseen, josta voisi yleistettynä olla hyötyä myös alkuperäisen ongelman ulkopuolella. Tässä opinnäytteessä pyritään nostamaan esiin juuri näitä tilanteita, joissa avoimen lähdekoodin sovelluksista löytyy ratkaisuja sellaisiin ongelmiin, joita ei ole aiemmin käsitelty tieteellisessä kirjallisuudessa. Näistä esimerkeistä tutkijat voivat saada Tichyn kaipaamia uusia ideoita uusien teorioidensa pohjaksi, jolloin ennen pitkää tiedon lisääntyessä ongelmiin saadaan entistä parempia ratkaisuja.

Tutkielman punaisena lankana eli tutkimusongelmana on erilaisia avoimen lähdekoodin sovelluksia tutkimalla ja vertailemalla selvittää, minkälaisia ongelmia niissä on kohdattu ja kuinka niitä on ratkottu. Tutkimuskohteeksi on valittu versionhallintaohjelmat, joten tutkimuksen käytännöllisessä osassa vertaillaan sitä, kuinka eri sovellusten yhteydessä on ratkottu versionhallintaan liittyviä ongelmia.

Tutkimuskohteen valinta selittyy sillä, että avoimen lähdekoodin viime vuosien suurimmat edistysaskeleet on otettu – Linux-ydin ja GNU-projekti poisluettuina – juuri versionhallinnan puolella. Erityisesti ns. BitKeeper-kriisin[4] jälkeen avoimen

[4] Tätä tapausta käsitellään tarkemmin luvun 2 kohdassa 2.15.1.

lähdekoodin versionhallintaohjelmien määrä on kasvanut nopeasti, jonka vuoksi aihe on nyt ajankohtainen: mistä tahansa avoimen lähdekoodin uutisia tarjoavasta palvelusta voi helposti löytää uutisia projekteista, jotka ovat vaihtaneet tai vaihtamassa käyttämäänsä versionhallintaohjelmaa. Kun tähän lisätään vielä yleinen kiinnostus avoimen lähdekoodin (ja vapaiden ohjelmien) projekteja kohtaan, voidaan todeta, että tutkimusongelma on sekä mielenkiintoinen, merkittävä että ajankohtainen.

1.4 Tutkielman rakenne

Aluksi luvussa 2 tutustutaan versionhallinnan historiaan ja esitellään aiheeseen liittyviä käsitteitä ja algoritmeja. Seuraavaksi luvussa 3 kuvataan vapaiden ohjelmien ja avoimen lähdekoodin käsitteiden kehittymistä. Sitä seuraavassa luvussa 4 perehdytään ensin tutkimusongelman teoreettisiin lähtökohtiin ja sen jälkeen katsotaan, kuinka versionhallinnan ongelmia on avoimen lähdekoodin puolella käytännössä ratkottu. Empiirisiä löydöksiä pohditaan teoreettisen tietämyksen valossa luvussa 5, jonka jälkeen kaikesta esitetystä tehdään yhteenveto luvussa 6.

Tämä tutkielma on tehty parityönä. Luvun 2 historiaosuus pohjautui alun perin Arin ja tekninen osuus Teron seminaariesitykseen. Myöhemmin tekstiä on lisätty, poistettu ja muokattu niin paljon, ettei mitään kohtaa voi enää varmistaa yksinomaan toisen tekemäksi.

Tekstin kirjoittaminen on tapahtunut iteratiivisesti. Toinen tekijöistä on yleensä hankkinut käsiteltävän aiheen taustatiedot ja kirjoittanut niiden avulla raakatekstin. Tämän jälkeen toinen on hankkinut samasta aiheesta lisäaineistoa ja varmentanut tämän avulla alkuperäisen tiedon. Samalla tekstiä on laajennettu ja kieliasua yhdenmukaistettu. Uuden iteraatiokierroksen alkaessa sama prosessi toistuu, tekijöiden vaihtaessa paikkaa.

Lopputuloksena syntyneen tekstin "tekijyyttä" on täten mahdotonta jakaa tekijöiden kesken, sortumatta minkäänlaiseen mielivaltaan. Tekijöiden mielestä paras mahdollinen jaottelutapa onkin seuraava: "Ari kirjoitti vokaalit ja Tero konsonantit. Välimerkit ja kieliasu ovat yhteisiä".

Lopuksi vielä muutama sana avoimuudesta. Tutkielmaa varten taustatietoja hankittaessa törmättiin samaan ongelmaan, jota harmitellaan mm. artikkelissa [30]: monet arvokkaat tieteelliset artikkelit on lukittu salasanojen taakse, joten niihin pääsee käsiksi vain yliopistojen tai muiden vastaavien tiedelaitosten kautta. Tämä on sel-

västi ristiriidassa tieteen avoimuuden vaatimuksen kanssa ja osoittaa sen, että ainakin tässä suhteessa virallisella tieteellä on vielä paljon opittavaa avoimen lähdekoodin toimintamalleista.

Hyvänä esimerkkinä avoimesta tiedonvälityksestä on suuren suosion saavuttanut Wikipedia. Vaikka sen luotettavuus monilta osin onkin vielä kyseenalainen, se on osoitus siitä, mitä yhteistyöllä voidaan saada aikaan. Tieteellisen tarkkaan ja luotettavaan Wikipediaan[5] on vielä matkaa, mutta sellaista odotellessa voimme kantaa oman kortemme kekoon antamalla tämän tutkielman vapaaseen käyttöön GFDL:n alla. 21. helmikuuta 2002 kirjoitimme suomenkieliseen Wikipediaan vain yhden lauseen[6]. Nyt voimme tarjota enemmän.

[5] Yksi yritys tällaiseen suuntaan on Citizendium, http://www.citizendium.org/ (viitattu 6.7.2007).

[6] http://fi.wikipedia.org/w/index.php?title=Wikipedia:Aikajana&oldid=2958287

2 Versionhallinta

> *Version control is the art of managing changes to information. It has long been a critical tool for programmers, who typically spend their time making small changes to software and then undoing those changes the next day. But the usefulness of version control software extends far beyond the bounds of the software development world. Anywhere you can find people using computers to manage information that changes often, there is room for version control.*
>
> *— Ben Collins-Sussman et al.[8]*

Ennen kuin versionhallinnasta voidaan tarkemmin puhua, on tiedettävä, mitä kyseisellä käsitteellä itse asiassa tarkoitetaan. Pelkkä sanalle "versionhallinta" annettava intuitiivinen merkitys kertoo toki sen, että tarkoituksena on hallita versioita, mutta siihen asia sitten oikeastaan jääkin. Mikä on versio ja kuinka sitä hallitaan?

Versionhallinnasta on annettu erilaisia määritelmiä eri aikoina. Seuraavassa joitakin määritelmiä eri vuosikymmeniltä (vapaasti suomennettuina):

- "Versionhallinta on toimintaa, jossa pyritään pitämään monista versioista ja konfiguraatioista koostuvat ohjelmistojärjestelmät hyvin organisoituina." (1985) [65]
- "Versionhallintajärjestelmä muodostuu joukosta työkaluja, joiden avulla seurataan ohjelmien ja sovellusten eri versioita ja konfiguraatioita." (1992) [31]
- "Versionhallinta tarjoaa mekanismin tuotteen komponenttien muutoshistorian seurantaan koko ohjelmistokehitysprojektin elinkaaren ajan. Sen avulla voidaan luoda erillisiä tuotekehitysvirtoja, joiden avulla voidaan ylläpitää erillisia versioita tuotteesta. Jokainen kehitysvirran sisältämä versio voidaan ottaa käyttöön. Jotkut järjestelmät tarjoavat mekanismin, jonka avulla kehitysvirran sisältämät muutokset voidaan tunnistaa ja yhdistää toiseen kehitysvirtaan." (1997) [45]
- "Nykyisen määritelmän mukaan konfiguraationhallinta tarkoittaa monimutkaisten järjestelmien evoluution hallintaa. Käytännössä se tarkoittaa sitä kurinalaista toimintatapaa, jonka avulla ohjelmistot voivat kehittyä hallitusti, ja mikä mahdollistaa laatu- ja aikavaatimusten saavuttamisen." (2000) [15]

- "Mahdollisuus tallentaa, palauttaa ja rekisteröidä alkion (dokumentin tai lähdekoodin) historiallinen kehitys on versionhallintajärjestelmän perimmäisin ominaisuus." (2001) [3]
- "Lähdekoodinhallinta on prosessi, jossa kontrolloidaan tai säädellään lähdekooditiedostojen uusien versioiden tuotantoa." (2001) [27]
- "Versionhallinta- ja konfiguraationhallintajärjestelmät mahdollistavat rinnakkaisten kehityslinjojen hallinnan ja aiempien versioiden ylläpidon." (2002) [33]
- "Versionhallinta on informaatioon kohdistuvien muutosten hallinnan taitoa." (2004) [8]
- "Versionhallintajärjestelmät ovat monen tietokeskeisen sovelluksen välttämätön osa. Pääsy mihin tahansa tiedon aiempaan tilaan mahdollistaa tiedon rakenteen tarkastelun millä tahansa hetkellä. Tästä on hyötyä silloin, kun halutaan palata aikaisempaan tilaan tai selvittää, miten data on kehittynyt nykytilaansa." (2006) [18]

Yhteisenä ideana näiden muodollisten määritelmien taustalla näyttäisi olevan se, että *versionhallinnan tehtävänä on pitää informaatioon tehtävät muutokset järjestyksessä sekä ajan että toisten muutosten suhteen*.

Koukeroisen kielen lisäksi kirjallisuudessa käytetään hyvin usein myös lyhenteitä, joiden merkitys saattaa vaihdella aina käyttäjästä riippuen. Tämä aiheuttaa ongelmia, ellei lyhenteitä aukaista niitä ensimmäistä kertaa käytettäesssä tai sisältö selviä muuten asiayhteydestä. Erityisen paljon sekaannuksia aiheuttaa lyhenne *SCM*, jolle on ainakin kaksi erilaista tulkintaa:

Source Code Management, lähdekoodin hallinta, tarkoittaa nimensä mukaisesti lähdekoodin hallintaa; yleensä lähdekoodiksi katsotaan varsinaisten koodirivejä sisältävien tiedostojen lisäksi myös mahdollinen dokumentaatio sekä kääntämiseen liittyvät metatiedostot (esim. Makefile ja configure.in).

Software Configuration Management, ohjelmistokonfiguraation hallinta, tarkoittaa kaiken ohjelmiston toimintaan vaikuttavan tiedon hallintaa; huomioitavia seikkoja ovat lähdekoodin lisäksi mm. kääntäjän ja käytettyjen kirjastojen versionumerot.

Tämä kahdella eri tavalla ymmärrettävä lyhenne aiheuttaa usein jonkinlaista sekaannusta, joten on suositeltavaa käyttää näissä eri tarkoituksissa eri lyhenteitä; CMS (Configuration Management System) ja VCS (Version Control System) ovat tähän hyviä vaihtoehtoja.

Tässä tutkielmassa lyhenne SCM esiintyy joidenkin vieraskielisten lainausten yhteydessä, jolloin sen merkitys käy ilmi asiayhteydestä. Muussa tekstissä pyritään selkeyden vuoksi välttämään turhia lyhenteitä, joten käytettävät suomenkieliset termit ovat yksiselitteisesti *konfiguraationhallinta* ja *versionhallinta*.

2.1 Versionhallinnan suhde konfiguraationhallintaan

Edellä on mainittu muutaman kerran käsite *konfiguraationhallinta*. Kuten aiemmasta voisi jo päätellä, tällekään käsitteelle ei ole saatu yleisesti hyväksyttyä määritelmää, joten sen sisältö vaihtelee tilanteesta riippuen. Dartin [12] mukaan konfiguraationhallinnasta voidaan puhua silloin, kun ainakin seuraavat tehtävät hoidetaan jollakin tasolla:

Identifiointi Identifiointimallilla kuvataan tuotteen rakenne. Se mahdollistaa tuotteen komponenttien käsittelyn nimeämällä komponentit ja niiden tyypit yksikäsitteisesti.

Kontrolli Julkaistujen tuotteiden ja niihin niiden elinkaaren aikana tehtyjen muutosten kontrollointi. Kontrollin tarkoituksena on pitää tuote yhtenäisenä ns. perusversion (engl. baseline) käytön avulla.

Tilatietojen ylläpito Komponenttien ja niihin liittyvien muutospyyntöjen hallinta ja raportointi, sekä komponenttien tilatietojen kerääminen ja ylläpito.

Auditointi Tuotteen kokonaisuuden ylläpito ja arviointi. Tarkoituksena on ylläpitää tuotteen eheyttä varmistamalla sen määrittelyn oikeellisuus ja komponenttien yhdenmukaisuus.

Kokoaminen Tuotteen valmistuksen ohjaus siten, että tarvittavat työvaiheet sujuvat parhaalla mahdollisella tavalla.

Prosessinhallinta Varmistetaan organisaation työtapojen ja -käytäntöjen sekä elinkaarimallien noudattaminen.

Ryhmätyö Tuotteen samanaikaisten käyttäjäryhmien työskentelyn ja kommunikaation koordinointi.

Näistä neljä ensimmäistä kuuluvat myös IEEE-standardiin 729-1983, ja loput ovat Dartin omia täydennyksiä. Millerin [46] jaottelun *tuotteen määrittely*, *versionhallinta*, *kokoamisenhallinta*, *muutostenhallinta* ja *laadunvarmistus* kattavat samat alueet, mutta tuovat mm. versionhallinnan osuuden selvemmin esiin.

Kuvassa 2.1 (s. 11) esitetään konfiguraationhallinnan toiminnalliset vaatimukset. Versionhallinta keskittyy kuvan kohtaan *Komponentit*.

Versionhallinta on konfiguraationhallinnan keskeinen osa-alue [71], joten sitä on tutkittu kauan. Frühaufin mukaan komponenttitason versionhallinta onkin mahdollisesti konfiguraationhallinnan parhaiten tunnettu osa-alue [19].

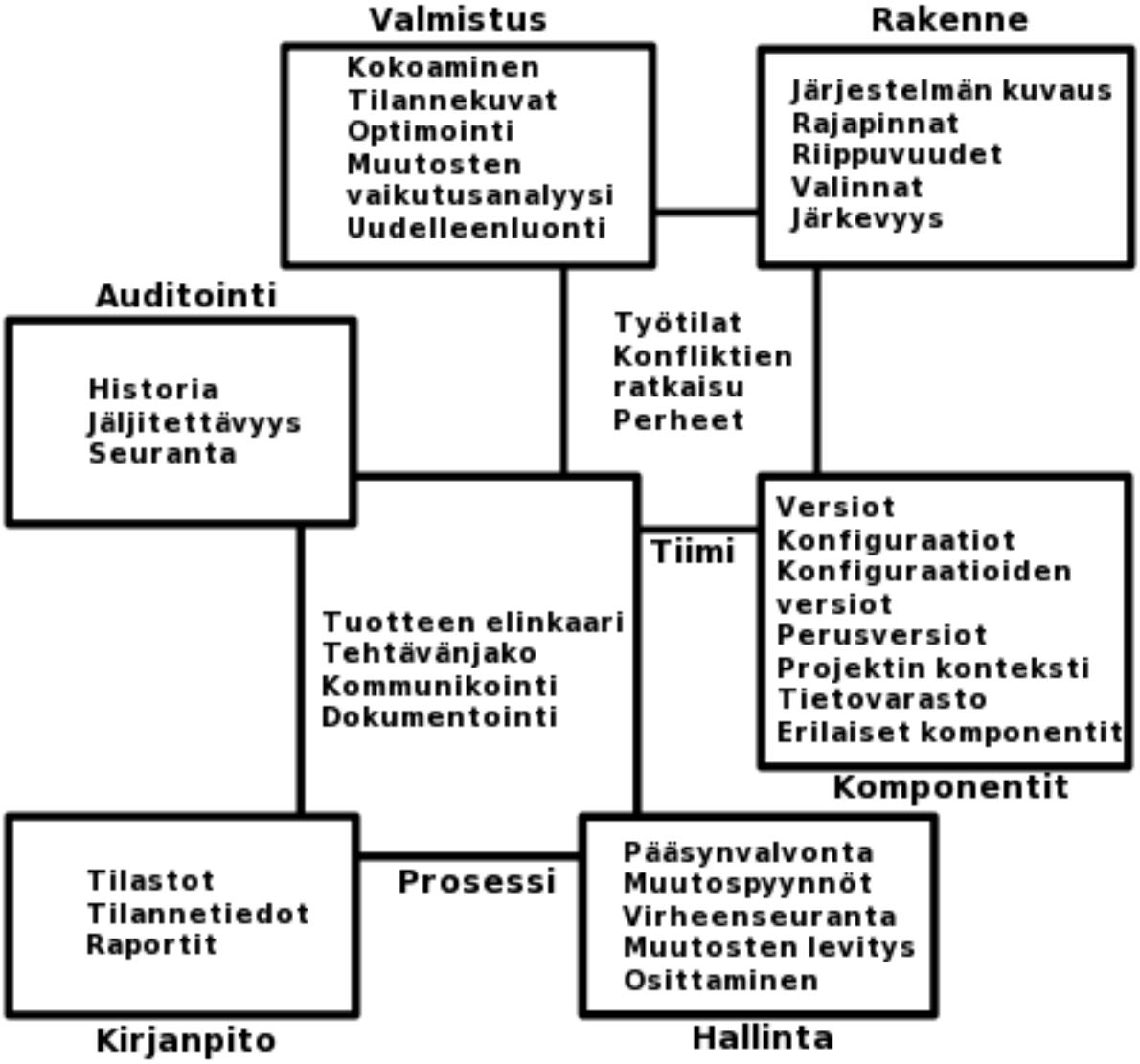
Seuraavaksi tehdään lyhyt katsaus versionhallinnan historiaan ja katsotaan, minkälaisia käsitteitä siihen liittyy ja kuinka ne ovat hiljalleen muotoutuneet. Vaikka versionhallinnan käsite onkin ollut olemassa jo yli kolmen vuosikymmenen ajan, sen periaatteet eivät ole juurikaan muuttuneet aikojen saatossa. Monia 1970-luvulta peräisin olevia ideoita, kuten deltoja, käytetään vielä nykyajan uusissakin järjestelmissä [15, 56], joten on hyvä tietää, kuinka nykytilaan on oikeastaan tultu.

2.2 Versionhallinnan tarve

Ohjelmistokehityksen aikana lähdekoodiin ja dokumentaatioon tehdään muutoksia. Kun tarvittavat muutokset on tehty, ne pitää tallentaa jotenkin. Yleensä kehittäjällä on tässä vaiheessa käytössään ohjelmiston sellainen versio, jossa muutokset ovat käytössä. Jäljellä on enää muutosten tallentaminen niin, että ne voidaan jakaa kaikille asianosaisille.

Aivan tietokoneaikakauden alussa versionhallinnalle ei kuitenkaan ollut tarvetta, koska käsiteltävää dataa säilytettiin tietokoneen ulkopuolella reikäkorteilla ja syötettiin koneeseen vain tarpeen mukaan. “Ohjelmistotkin” olivat vain joukko sähköisiä kytkentöjä, ja siten ainutlaatuisia: vanhoja versioita ei jäänyt, koska jokainen tehty muutos korvasi vanhan toiminnan uudella. [65]

Tekniikan kehittyessä tilanne vähitellen muuttui: ohjelmoinnissa siirryttiin fyysisten kytkentöjen suunnittelusta loogisten operaatioiden hallintaan ja käsiteltävä data siirtyi reikäkorteilta massamuistiin. Suoritettavat ohjelmat ja käsiteltävä data voitiin esittää samassa muodossa, eikä niiden välillä ollut enää selvää erotusta – nyt ohjelmiakin pystyi versioimaan. [65]



Kuva 2.1: Konfiguraationhallinnan osa-alueet.

Tallennuskapasiteetin kasvaessa tuli mahdolliseksi säilyttää käsiteltävästä datasta useita eri versioita, mikä mahdollisti alkeellisen versionhallinnan: jos uusimmassa versiossa havaittiin jotain vikaa, oli helppo palata aiemmin toimineeseen versioon. [15, 65]

2.3 Alkeellisia versionhallintamenetelmiä

Ensimmäinen alkeellinen "versionhallintamenetelmä" oli tallentaa käsiteltävän datan eri versiot eri tiedostoihin; alun perin näitä tiedostoja ei nimetty kovinkaan järjestelmällisesti, vaan nimet olivat tyyliä `raportti` ja `uudempi_raportti`[1]. Käyttäjiä ajatellen tämä nimeämiskäytäntö oli varsin toimiva, mutta tietokoneen kannalta ratkaisu ei ollut riittävän järjestelmällinen, joten sitä ei voitu automatisoida. Menetelmä osoittautui myös epäkäytännölliseksi tilanteissa, joissa käsiteltiin tavanomaista laajempia sovelluksia: useista lähdekooditiedostoista koostuvan projektin tiedostoversioiden hallinta kävi liian työlääksi, joten asiaan piti saada jokin muu ratkaisu.

Ratkaisu löytyi ns. *versionumero*n käytöstä. Kun muutettavien tiedostojen nimiin alettiin lisätä versiota ilmaiseva juokseva numero, jota kasvatettiin aina uuden version yhteydessä, niin tietokonekin pystyi tunnistamaan, että tiedosto `raportti.1` on `raportti.2`:n vanhempi versio. Tämä menetelmä oli oivallinen myös suurille tiedostojoukoille, etenkin silloin, kun jotkin tietokonejärjestelmät, kuten VMS[2][11, 65], alkoivat tukea tiedostojen automaattista versiointia. Tällöin käyttäjiltä ei vaadittu lainkaan lisätyötä, joten he saivat keskittyä kokonaan muutosten tekoon.

2.4 Tallennustilan minimointi

Useiden versioiden säilyttäminen kulutti kuitenkin arvokasta (ja kallista) tallennustilaa: pienenkin muutoksen aiheuttama versionumeron kasvu vaati tallentamaan koko dokumentin uudelleen. Yleiseksi tavaksi muodostuikin säännöllisesti poistaa vanhimpia versioita ja pitää tallessa vain muutamia uusimpia. Tämä menettely kyl-

[1] Todellisuudessa tiedostojärjestelmä kuitenkin rajoitti nimeämismahdollisuuksia: esimerkiksi Forth-kielen nimen piti alun perin olla Fourth, mutta Charles Mooren käyttämän tietokoneen tiedostojärjestelmä rajoitti tiedostonimet viiteen merkkiin. (http://www.forth.com/resources/evolution/evolve_0.html, viitattu 23.7.2007.)

[2] Virtual Memory System

lä toimi, mutta siinä oli yksi heikkous: vanhimmat versiot hävisivät lopullisesti, joten niitä ei voinut enää käyttää. [57]

2.4.1 SCCS

Ratkaistakseen tämän ongelman, Marc J. Rochkind alkoi vuonna 1972 kehittää uudentyyppistä, *limittäisiin deltoihin* (“weave”) perustuvaa järjestelmää. Kehityksen tuloksena syntyi *SCCS* (Source Code Control System), joka esiteltiin tieteellisessä artikkelissa [57] vuonna 1975 ja jota voidaan pitää ensimmäisenä varsinaisena versionhallintaohjelmistona. [43, 57]

SCCS:n lähtökohtana oli havainto, jonka mukaan tiedostojen peräkkäiset versiot poikkeavat toisistaan yleensä hyvin vähän. Tällöin eri versioiden tallentaminen voidaan tehdä tehokkaasti siten, että jokaisesta tiedostosta tallennetaan kokonaisena vain ensimmäinen versio, jolloin myöhemmät versiot saadaan talteen säilyttämällä erikseen näiden *perusversioiden* (engl. base version) päälle tehdyt muutokset. Näitä muutoksia nimitetään matematiikasta lainatun termin mukaisesti *deltoiksi*. [15, 57]

Deltojen käyttö tuo merkittävän tilansäästön. Estublierin [15] mukaan tiedoston kaksi peräkkäistä versiota ovat keskimäärin jopa 98-prosenttisesti identtiset, jolloin niiden välinen delta jää vähäiseksi. Tätä väitettä tukevat myös Conradi ja Westfechtel [10], joiden mukaan sopivia deltamenetelmiä käyttämällä eri versioiden tilankäyttö saadaan putoamaan murto-osaan (2–3%) alkuperäisestä. Aikaisemmassa artikkelissaan he kuitenkin muistuttavat, etteivät deltat suinkaan aina jää pieniksi, sillä ne riippuvat suuresti käsiteltävien tiedostojen ominaisuuksista. [9]

2.4.2 RCS

Deltoja käytettiin myös Walter F. Tichyn vuonna 1982 kehittämässä *RCS*-ohjelmassa (Revision Control System), joka esiteltiin tieteellisesti [65] vuonna 1985. Se oli periaatteiltaan samanlainen kuin SCCS, mutta sisäisesti hieman erilainen. Kun SCCS käytti limittäisiä deltoja, niin RCS käytti *takautuvia deltoja* (“reverse-delta”). Tämä nopeutti jonkin verran uusimpien versioiden muokkausta, koska ne säilytettiin kokonaisina.

Aikojen kuluessa deltamenetelmät ovat muuttuneet, mutta perusideat ovat pysyneet samoina. Seuraavassa katsotaan hieman tarkemmin, minkälaisia deltoja oikeastaan on käytössä ja miten ne toimivat.

2.4.3 Deltat

Limittäisiä deltoja (engl. weave) käytetään mm. SCCS:ssä ja BitKeeperissä. Niissä muutokset säilytetään yhdessä tiedostossa siten, että jokaiseen versioon kuuluva osuus on selkeästi merkitty. Idea selviää seuraavasta pseudokoodista [17, 57]:

```
#if V1 || V2
Ensimmäinen rivi
#endif
#if V2
Toinen rivi
#endif
#if V3
Ainoa rivi
#endif
```

Tässä tiedoston ensimmäinen versio sisältää tekstin `Ensimmäinen rivi`, versio 2 rivit `Ensimmäinen rivi` ja `Toinen rivi` ja kolmas versio pelkästään tekstin `Ainoa rivi`. Toimintatapa on siis samankaltainen kuin esim. C-kielen esikääntäjällä, vaikkakin todellisuudessa tiedoston muoto ei ole näin yksinkertainen. [43, 57]

Eteneviä deltoja (engl. forward delta) käytettäessä tehdyt muutokset tallennetaan perusversion päälle siten, että uudempaan versioon päästään lisäämälle deltoja aiempien versioiden päälle; esim. `V2 = V1 + D2` ja `V4 = V1 + D2 + D3 + D4`.

Takautuvien deltojen (engl. reverse delta) periaate on samanlainen, mutta perusversiona käytetään aina uusinta versiota, jolloin deltoja tarvitaan vain aikaisempaan versioon siirryttäessä: esim. `V1 = V4 + D3 + D2 + D1`, jossa jokainen delta siis pudottaa versiota yhdellä.

Algoritmiselta kannalta tarkasteltuna kaikki edellä mainitut tekniikat ovat aikavaativuudeltaan lineaarisia (weave tiedoston koon ja muut muutosten määrän suhteen) [49]; versiosta toiseen siirryttäessä joudutaan käymään läpi kaikki deltat, jotka sijoittuvat nykyversion ja halutun version väliin. Tilanne on siis samantapainen kuin linkitetyn listan läpikäynti.

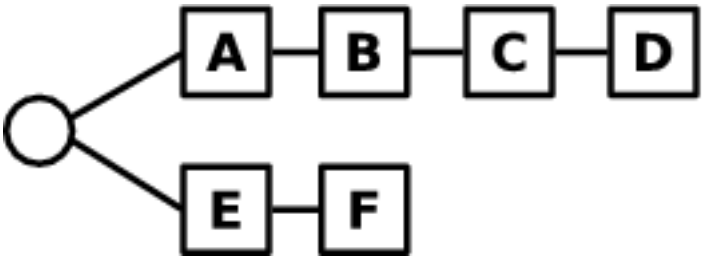
Myöhemmin katsotaan, minkälaisia parannuksia näihin perinteisiin deltamenetelmiin on kehitetty, sekä kuinka näitä deltoja luodaan, eli minkälaisilla algoritmeilla tiedostojen väliset erot saadaan selville. Tätä ennen on kuitenkin hyvä selvittää lyhyesti joitakin asiaan kuuluvia käsitteitä.

2.5 Versio, revisio ja variantti

Conradi ja Westfechtel [9] määrittelevät version olevan kehittyvän alkion tilan tiettynä hetkenä. Alkio voi puolestaan olla mikä tahansa versionhallintaan laitettava asia, kuten tiedosto, hakemisto tai oliotietokannan olio. Buffenbargerin ja Gruellin antama määritelmä on tämän kanssa yhtenevä [5].

Versio on siis jonkin alkion tila tiettynä hetkenä. Tämä määritelmä ei ole yksinään aivan riittävä, joten otetaan käyttöön uusi käsite, joka ilmaisee eri versioiden suhdetta toisiinsa: *revisio* on alkion sellainen versio, jonka tarkoituksena on korvata edeltäjänsä [71]. Uusi revisio voi siis syntyä esimerkiksi bugikorjauksen vuoksi, jolloin vanhempaa versiota ei pitäisi sen jälkeen enää käyttää.

Version ja revision lisäksi käytössä on vielä kolmas käsite: *variantti* on alkion sellainen versio, jonka on tarkoitus olla olemassa samanaikaisesti muiden versioiden kanssa [71]. Esimerkiksi jostakin tietorakenteesta voi olla käytössä useita toteutuksia, jotka eroavat toisistaan resurssienkäytön suhteen.



Kuva 2.2: Versio, revisio ja variantti.

Asia on ehkä helpointa ymmärtää esimerkin avulla. Kuvassa 2.2 A, B, C, D, E ja F ovat kaikki jonkin alkion eri versioita. A, B, C ja D ovat toistensa revisioita, koska uusi versio korvaa edellisen. E ja F ovat puolestaan edellisten variantteja ja toistensa revisioita.

Seuraavaksi tarkastellaan sitä, kuinka näiden olioiden suhteet toisiinsa saadaan pysymään järjestyksessä.

2.6 Puut ja graafit

Conradi [9] ja Westfechtel [71] toteavat, että erilaisten versiointimallien kuvaaminen onnistuu graafeilla. Graafit muodostuvat solmuista (engl. node) ja kaarista (engl. edge). Solmut kuvaavat alkioiden versioita ja kaaret niiden suhteita toisiinsa. Kaa-

ret ovat suunnattuja siten, että kahden peräkkäisen version välinen kaari osoittaa vanhemmasta versiosta uudempaan.

Versionhallintaohjelma voi pitää versioidut alkiot järjestyksessä usealla tavalla. Yksinkertaisimmillaan versiot muodostavat lineaarisen versiograafin, mikä tarkoittaa yksinkertaisesti sitä, että uudet versiot laitetaan aina edellisten versioiden perään. [3] Näin muodostuva ketju on melko rajoittunut, mutta sopii pienimuotoiseen versionhallintaan. Peräkkäiset versiot ovat toistensa revisioita.

Lineaarisen versiograafin ongelmana on, ettei se salli vanhempien versioiden muokkausta. Tällainen tilanne tulee kuitenkin käytännössä vastaan varsin usein. Jos esimerkiksi julkaistusta ohjelmasta löytyy jokin virhe, on tavallista, että se korvataan uudemmalla korjatulla versiolla, joka perustuu aiemmin julkaistun tuotteen koodiin. Mikäli kehitystä on kuitenkin jatkettu alkuperäisen julkaisun jälkeen, tarvittavia korjauksia ei voi tehdä vanhempaan versioon, vaan ne on tehtävä aina uusimman version päälle. Toinen lineaarisen versiograafin ongelma on, ettei se mahdollista useamman kehittäjän samanaikaista työskentelyä. Tästä johtuen myöskään eri varianttien käsittely ei onnistu.

Lineaarista versiograafia parempi keino on käyttää puumaista rakennetta. Yksinkertaisimmillaan puurakenne voi olla täysin lineaarinen, mutta lineaarisesta graafista poiketen siinä versiohistoria voi *haaroittua* eli muodostaa graafiin sellaisia solmuja, joiden jälkeen tulee kaksi tai useampia rinnakkaisia versioita. Näillä rinnakkaisilla versioilla on sama esi-isä, joten ne ovat toistensa variantteja.

Puumaisen rakenteen etuna on, että se mahdollistaa aiemmin kuvatun vanhemman version muokkauksen ja rinnakkaisen kehityksen. Tällaisen graafin heikkoutena puolestaan on se, että haaroittuneen versiohistorian haarat ovat täysin erillisiä. Niiden ainoa yhteinen ominaisuus on, että niillä on yhteinen esi-isä, eli se versio, jonka kohdalla haarautuminen on tapahtunut.

Uusien ominaisuuksien kehittely eri haaroissa on järkevää, sillä eristettyinä ne eivät pääse sotkemaan toisiaan. Kehitystyön valmistuessa eri haarojen sisältämät toiminnot on kuitenkin saatava siirrettyä siihen päähaaraan, josta sovelluksen lopullinen versio julkaistaan. Puumaista versiograafia käytettäessä tämä ei kuitenkaan onnistu, sillä siinä kehityshaarat jäävät aina erillisiksi.

Tilanne korjaantuu ottamalla käyttöön *suunnatut syklittömät graafit* (engl. directed acyclic graphs, DAG). Niissä eri versioiden suhde toisiinsa voi olla lähes mikä tahansa, kunhan graafissa ei ole silmukoita. Silmukoiden esiintyminen johtaisi siihen epäloogisuuteen, että jokin versio olisi itse oma esi-isänsä. Tavallisesti graafien kaik-

kia mahdollisuuksia ei kuitenkaan käytetä, vaan niitä käsitellään eräänlaisina laajennettuina puina siten, että eri haarat voidaan palauttaa yhdeksi haaraksi. Tällöin yhdistyskohtaan tulee solmu, jolla on useita esi-isiä.

2.7 Tietovarastot

Vuonna 1986 Dick Grune ja kaksi hänen oppilastaan kehittivät RCS:n päälle kasan shelliskriptejä, joiden avulla he pystyivät kehittämään projektinsa osia samanaikaisesti. [23]

CVS poikkesi aiemmista versionhallintaohjelmista siinä, että se käsitteli koko hakemistoa yksittäisten tiedostojen sijaan. Käytössä oli erillinen *tietovarasto* (engl. repository), jonne kaikki versionhallintatiedot tallennettiin. Käyttäjät hakivat tiedostot tietovarastosta omiin työhakemistoihinsa, joissa kaikki kehitys ja muokkaus tapahtui.

Tietovarasto voi olla lähes mikä tahansa tiedon tallentamiseen sopiva paikka, kuten yksittäinen tiedosto, hakemisto tai vaikkapa tietokanta. Käyttäjän kannalta toteutustavalla ei ole juurikaan merkitystä, mutta on hyvä tietää, että eri tavat vaikuttavat mm. järjestelmän nopeuteen ja tilankäyttöön, tietoturvasta puhumattakaan. Esimerkkeinä erilaisia tietovarastoja käyttävistä sovelluksista voidaan pitää vaikkapa Monotonea ja Subversionia, joista edellinen tallentaa tiedot SQLite-nimiseen tietokantaan [29] ja jälkimmäinen joko Berkeley DB -tietokantaan tai uudemmalle virtuaaliselle FSFS-tiedostojärjestelmälleen. [8]

Tietovarasto voi olla paikallinen tai sijaita jossain muualla, ja se voi olla joko keskitetty tai hajautettu. Sen päätehtävänä on eri versioiden säilytys ja niihin pääsyn mahdollistaminen. Tämän lisäksi tietovarasto ylläpitää versioiden muutoksiin liittyviä muutostietoja, kuten tietoja muutosten tekijöistä ja ajankohdista. Joskus tietovarasto voi sisältää pääsynhallintatietoja, jolloin vain tietyt kehittäjät pääsevät käsiksi sen sisältämiin tietoihin. [8]

Paikallinen tietovarasto sijaitsee nimensä mukaisesti yleensä työaseman kiintolevyllä tai paikallisesti näkyvällä verkkolevyllä, joten sitä voi käyttää pääsääntöisesti ilman yhteyttä ulkoiseen verkkoon. Esimerkkejä tällaisia tietovarastoja käyttävistä sovelluksista ovat mm. SCCS ja RCS. Näiden lisäksi myös tietoverkkoja hyödyntävät ohjelmat osaavat toimia paikallisten tietovarastojen kanssa, mutta tällöin monet niiden ominaisuuksista jäävät hyödyntämättä.

Paikalliset tietovarastot sopivat hyvin pieniin projekteihin, joissa kehittäjien vä-

liseen tiedonsiirtoon ei ole suurta tarvetta:

+ Tiedot ovat aina saatavilla, ei tarvetta verkkoyhteydelle.

+ Tietovaraston pääsynvalvonta on helppo toteuttaa järjestelmän oikeuksienhallinnan avulla.

− Muutosten jakelu vaatii lisätyötä.

− Ulkopuoliset muutokset pitää ottaa mukaan paikallisesti.

− Tietovaraston varmuuskopiointi voi kuulua käyttäjän tehtäviin.

Keskitetyt tietovarastot sijaitsevat lähes aina verkossa. Niitä käytettäessä kehittäjät saavat yhteisestä tietovarastosta käyttöönsä joko paikallisen kopion tai näkymän, jonka avulla tietoja voidaan muokata. Kopiot ovat yleensä osin tai kokonaan riippuvaisia alkuperästään. Esimerkiksi CVS:n tapauksessa edes paikallisia muutoksia ei voi nähdä ilman verkkoyhteyttä, sillä niiden selvittäminen vaatii paikallisen kopion vertaamista alkuperäiseen tietovarastoon. Näkymä puolestaan voidaan toteuttaa esimerkiksi paikallisesti näkyvän verkkolevyn tai versionhallintaohjelman oman mekanismin avulla. Näkymää käytettäessä tiedostojen ei tarvitse sijaita fyysisesti käyttäjän tietokoneella.

Keskitetyt tietovarastot mahdollistavat hyvät seuranta- ja hallintamahdollisuudet, mutta vaativat myös asianmukaista ylläpitoa:

+ Tietojen varmuuskopiointi on helppoa.

+ Pääsy tietovarastoon voidaan estää tai sallia käyttäjäkohtaisesti.

+ Tehdyistä muutoksista voidaan lähettää automaattinen ilmoitus kaikille asianosaisille.

+ Muutoksille voidaan asettaa tarkistuksia tallennuksen yhteydessä (esim. tekijänoikeusmerkintä tiedoston alussa[3]).

− Sujuva toiminta vaatii asianmukaisen ylläpidon.

[3]Tekijänoikeutta suojaava Bernin yleissopimus tuli Yhdysvalloissa voimaan vasta ns. “Berne Convention Implementation Act of 1988” -lain nojalla vuonna 1989; sitä ennen ©-merkintä oli välttämätön tekijänoikeuden osoittamiseksi.

Hajautetussa tapauksessa jokainen alkuperäisestä tietovarastosta tehty kopio on itsenäinen ja tasavertainen. Tämä tarkoittaa sitä, että tietoja voidaan siirtää myös eri kopioiden välillä, jolloin alkuperäisellä tietovarastolla ei ole erityisasemaa.

Hajautetuissa tietovarastoissa yhdistyvät valikoidusti molempien edellä mainittujen tyyppien sekä hyvät että huonot puolet, sillä niiden käytössä voidaan poimia tapoja kummastakin:

+ Verkkoyhteyttä voidaan hyödyntää, mutta sitä ei tarvita.

+ Muutosten jakelu on helppoa.

± Ylläpidon määrä riippuu käyttötavoista.

− Pääsynvalvonta pitää konfiguroida tilannekohtaisesti.

Versionhallinnan alkuaikoina tietovarastot olivat poikkeuksetta paikallisia, mutta tietoverkkojen kehittyessä niiden mahdollisuuksia alettiin hyödyntää ottamalla käyttöön verkossa sijaitsevat keskitetyt tietovarastot. Uudemmissa sovelluksissa tämä kehitys on edelleen jatkunut, joten nykyään eletään hajautettujen tietovarastojen valtakautta.

2.8 Kehityshaarat

Kehityshaarat (engl. branch) ovat suoraviivaisen kehityshistorian laajennos. Tavallisesti uusin versio korvaa aina edellisen version, eli versiot muodostavat eräänlaisen ketjun ensimmäisestä versiosta viimeisimpään. Kehityshaaran tapauksessa jonkin version jälkeen versioketju voi haarautua, eli sen jälkeen voi olla kaksi siitä muodostettua versiota, joiden kummankin edellinen versio on alkuperäinen versio. [8]

Kuvan 2.2 (s. 15) esittämässä tapauksessa alkuperäinen versio on haarautunut kahdeksi kehityshaaraksi. Ylemmässä haarassa on tehty neljä muutosta ja alemmassa kaksi muutosta.

Useimmat versionhallintaohjelmat tukevat kehityshaarojen käyttöä, mutta niiden toteutustapa vaihtelee melkoisesti. Esimerkiksi GITissä kaikki kehityshaarat ovat samassa tietovarastossa, ja ne tunnistetaan 40-merkkisen tunnisteen avulla. Subversionissa haarat puolestaan tallennetaan tietovarastoon erillisinä alihakemistoina.

Tavallisesti haarautumista ei rajoiteta mitenkään, eli jokin haara voi puolestaan haarautua uudestaan, jonka jälkeen uusi haara haarautuu myös jne. Vähitellen versiohistoria muotoutuu puumaiseksi.

Kehityshaaran etuna on se, että se mahdollistaa alkuperäisen version samanaikaisen kehityksen erillisissä kehityshaaroissa. Erilliset muutokset voivat aiheuttaa ongelmia, mutta niihin on kehitelty erilaisia ratkaisukeinoja.

2.9 Lukitukset

Versionhallinnan alkuaikoina yleisenä käytäntönä oli, että käsiteltävä data lukittiin käsittelyn ajaksi; ilman lukitusta kaksi samanaikaista muutosta olisi voinut sekoittaa toisensa ja aiheuttaa odottamattomia seurauksia. Käytäntö oli siis tarpeellinen, mutta sillä oli omat haittapuolensa: jos yhtäaikaisia käyttäjiä oli useita, lukituksista saattoi muodostua kaikkia hidastava pullonkaula.

2.9.1 Lukkotiedostot

Perinteisissä järjestelmissä, kuten RCS, käytettiin yhteisymmärrykseen perustuvia lukituksia, eli ns. *lukitse-muokkaa-vapauta*-menetelmää (engl. lock-modify-unlock). Siinä kulloinkin dataa käsittelevä sovellus loi ns. *lukkotiedoston*, jonka perusteella muut sovellukset ymmärsivät pidättäytyä muutosten teosta. Kun operaatio saatiin valmiiksi, lukkotiedosto poistettiin ja käsittelyvuoro siirtyi sille, joka ehti tehdä uuden lukituksen ensimmäisenä; muut saivat jäädä odottamaan uutta tilaisuutta. [8, 15, 17]

Vaikka lukitusten käyttö onkin yleensä tarpeellista, siihen liittyy joitakin ongelmia, joista Ben Collins-Sussman, Brian W. Fitzpatrick ja C. Michael Pilato nostavat esille seuraavat kolme skaalautuvuuteen liittyvää kohtaa [8]:

1. Lukitusten käyttö voi aiheuttaa ylläpito-ongelmia. Jos esimerkiksi kehittäjä A unohtaa lukinneensa jonkin tiedoston, kehittäjä B ei voi muokata samaa tiedostoa, ennen kuin lukitus avataan. Mikäli A lähtee lomalle, voi kulua pitkä aika, ennen kuin B huomaa lukituksen jääneen vahingossa päälle ja pyytää ylläpitäjää poistamaan sen. Tarpeettomasta lukituksesta käytetään nimitystä *stale lock*.

2. Lukitusten käyttö voi pakottaa peräkkäiseen työskentelyyn, vaikka rinnakkainenkin työskentely olisi mahdollista. Jos esimerkiksi A muokkaa tiedoston al-

kuosaa ja B loppuosaa, kummankaan muutokset eivät aiheuta konflikteja. Mikäli tehdyt muutokset voidaan yhdistää oikein, ei ole mitään syytä pakottaa toista kehittäjää odottamaan, kunnes toinen on saanut tehtävänsä valmiiksi.

3. Lukitusten käyttö voi johtaa väärään turvallisuudentunteeseen. Kun esimerkiksi A lukitsee jonkin tiedoston, hän voi pitää itsestään selvänä sitä, että hänen muutoksensa ovat toisten kehittäjien muutoksista eristettyjä. Tämän vuoksi hän ei pidä tarpeellisena kertoa muille, mitä hän oikeastaan aikoo tehdä. Samanaikaisesti B voi muokata jotain toista tiedostoa, joka riippuu A:n muokkaamasta tiedostosta. Myös B uskoo tekevänsä omia muutoksiaan muista muutoksista eristettynä, joten hänkään ei ole maininnut kenellekään, mitä on tekemässä. Jos molemmat muokattavat tiedostot riippuvat toisistaan, voi käydä niin, että tehdyt muutokset ovat epäyhteensopivia. Tämä aiheuttaa parhaassa tapauksessa virheen ohjelmaa käännettäessä, mutta pahimmassa tapauksessa ohjelma kääntyy oikein, mutta toimii virheellisesti. Kummassakin tapauksessa joudutaan tekemään turhaa työtä ongelman korjaamiseksi.

Nämä kohdat ovat varsin ongelmallisia etenkin hajautetuissa ympäristöissä, joissa kehittäjiä voi olla useita kymmeniä. Lukitusten aiheuttamien skaalautuvuusongelmien vuoksi asioita onkin jouduttu suunnittelemaan uusiksi, jolloin on päädytty varsin mielenkiintoiseen tulokseen: lukitusten käyttö ei olekaan aina välttämätöntä.

2.9.2 Lukitusten välttäminen

Eräs keino välttää turhia lukituksia on mm. CVS:n ja Subversionin käyttämä optimistinen *kopioi-muokkaa-yhdistä*-menetelmä (engl. copy-modify-merge), jonka nimi on varsin kuvaava. Tässä menetelmässä kehittäjä kopioi tietovaraston senhetkisen tilan omaan työhakemistoonsa, eikä lukituksia tämän lyhyen kopiointiprosessin jälkeen enää tarvita. Kopioinnin jälkeen kehittäjä voi muokata tiedostoja vapaasti kenenkään häiritsemättä. Kun tehdyt muutokset halutaan myöhemmin yhdistää tietovarastossa olevaan versioon, versionhallintaohjelma huomaa, mikäli niihin on tehty sellaisia muutoksia, jotka pitäisi ottaa huomioon. [3, 8]

Mikäli konfliktoituvia muutoksia on tullut, kehittäjän on selviteltävä ristiriidat, ennen kuin hän voi tallentaa omat muutoksensa. Tavallisesti tämä on kuitenkin helppoa, koska tarkastelua vaativat kohdat ovat juuri niitä, joita kehittäjä on äskettäin käsitellyt.

Collins-Sussman ym. [8] toteaa, että näennäisestä kaoottisuudestaan huolimatta optimistinen lukitusmalli toimii käytännössä erittäin hyvin. Käyttäjät voivat työskennellä samanaikaisesti ja jopa muokata samoja tiedostoja. Kaikki konfliktit tulevat vain sellaisiin kohtiin, jotka kehittäjä tuntee entuudestaan. Konfliktien ratkaisuun kuluva aika on vähäinen verrattuna odotteluun, jota lukitusten käyttö aiheuttaisi.

Lukitusten välttäminen on hyvä tapa silloin, kun se toimii. On kuitenkin tapauksia, joissa lukitukset ovat välttämättömiä. Esimerkiksi binääritiedostojen yhdistäminen on ainakin toistaiseksi hyvin vaikeaa tai jopa mahdotonta. Mikäli jo muokkausta aloitettaessa tiedetään, ettei samanaikaisia muutoksia voi yhdistää, on järkevää lukita tiedosto muokkauksen ajaksi.

Uudemmissa järjestelmissä lukitusten aiheuttamia viiveitä ja ongelmia on vähennetty yksinkertaisella tavalla: lukituksia ei tehdä ollenkaan!

Käytettävä menetelmä on yksinkertainen ja perustuu tiedoston nimeämiseen sen sisällön perusteella. Esimerkiksi GITissä ja Mercurialissa (ja niitä inspiroineessa Monotonessa) tiedostot nimetään niistä laskettujen SHA-1-arvojen (s. 191) mukaisesti. Tällöin käytännössä jokainen muutos muuttaa tiedoston nimeä, eli jokainen muutos tallentuu erilliseen tiedostoon. Merkittävää tässä on se, että usea käyttäjä saa muuttaa samaa tiedostoa samanaikaisesti: muutokset eivät mene päällekkäin, ja jos menevät, niin lopputulos on kuitenkin (todennäköisesti[4]) oikea. [24, 29, 43]

2.10 Muutosten havaitseminen

Lähdekoodi ei tule milloinkaan valmiiksi, sillä sitä voidaan aina muuttaa. Yleisimpiä syitä muutosten tekoon ovat virheiden korjaaminen ja uusien ominaisuuksien lisäys.

Tehdyt muutokset voivat rajoittua paikallisesti pienelle alueelle yksittäiseen tiedostoon, tai ne voivat levittäytyä useaan paikkaan monen tiedoston sisälle. Tällöin voi olla hankalaa nähdä, mitä oikeastaan tehtiin, joten tehtäväksi tulee löytää jonkinlainen keino tehtyjen muutosten havaitsemiseen. Periaatteessa olisi helppo verrata alkuperäistä ja muutettua versiota toisiinsa, mutta käytännössä tämä on vaikeaa. Tästä syystä on kehitetty erilaisia algoritmeja, joiden avulla muutokset voidaan paikallistaa helpommin.

Muutosten automaattisella havaitsemisella on suuri merkitys sovelluskehityk-

[4] Virheen todennäköisyys on 1 / 1208925819614629174706176.

seen. Kun tehtyjä muutoksia voi jälkikäteen tarkastella muusta lähdekoodista erikseen, niiden oikeellisuuden tarkistaminen helpottuu huomattavasti. Tämä helpottaa myös virheenetsintää, sillä ongelmallinen kohta löytyy yleensä viimeksi tehtyjen muutosten joukosta.

2.10.1 Muutosten havaitsemisen teoreettinen perusta

Muutosten havaitsemiseen käytetyt keinot pohjautuvat 1950-luvun tutkimuksiin virheenhavaitsevista ja virheenkorjaavista koodeista. Richard Hammingin esittelemän *Hamming-etäisyyden* avulla voitiin laskea yksittäisten bittien muutoksia. [25]

Vuonna 1966 Vladimir Levenshtein esitteli artikkelissaan [38] Hamming-etäisyyden yleistyksen. Kun Hamming-etäisyyden laskemisessa muutokseksi katsottiin bitin vaihtuminen toiseksi, *Levenshteinin etäisyydessä* bittien sijaan käytetään merkkejä, ja muutokset voivat olla merkkien korvaamisen lisäksi yksittäisten merkkien lisäyksiä ja poistoja.

Levenshteinin etäisyyden avulla voitiin laskea ns. *editointietäisyys* eli se määrä operaatioita, jolla merkkijono saadaan muutettua toiseksi. Editointietäisyyttä voidaan käyttää esimerkiksi kirjoitusvirheiden havaitsemiseen.

Wagnerin ja Fischerin artikkeli "The String-to-String Correction Problem" [69] esitteli ongelman, jossa tavoitteena on selvittää kahden merkkijonon etäisyys toisistaan. Etäisyydellä tarkoitettiin pienintä muutosten määrää, jolla merkkijonot saatiin muokattua samanlaisiksi. Muutosoperaatioiksi katsottiin samat operaatiot, kuin Levenshteinin etäisyyden tapauksessa. Artikkelissa osoitettiin myös, miten merkkijonojen pisin yhteinen osajono (engl. Longest Common Subsequence, LCS) voidaan löytää[5].

Näiden algoritmien pohjalta syntyi useita merkittäviä sovelluksia, kuten Unix-järjestelmien diff-ohjelma [32], James Goslingin näytönpäivitysalgoritmi [21] ja Tichyn string-to-string correction -algoritmiin kehittämä lohkonsiirtomalli (engl. block-moves model) [64], jota käytettiin myöhemmin RCS-ohjelmassa. Vuonna 1986 Myers [47] osoitti, että pisimmän yhteisen osajonon etsintä vastaa lyhimmän polun etsintää editointigraafissa. Tätä havaintoa voidaan hyödyntää mm. Diff-algoritmissa.

[5] Pisin yhteinen osajono on sellainen jono, joka on molempien jonojen osajonoina. Jonon merkkien ei tarvitse olla peräkkäin, joten esimerkiksi sanojen KISSA ja KOIRA pisin yhteinen osajono on KIA.

2.10.2 Diff-algoritmi

Diff-algoritmin lähtökohtana on idea siitä, että mikä tahansa merkkijono voidaan muuttaa toiseksi tekemällä siihen pieniä yksittäisiä muutoksia, esimerkiksi lisäämällä tai poistamalla yksittäinen kirjain. [32] Alku- ja lopputilan eroavuudet voidaan siten esittää ns. *muutostiedostona* (engl. delta file), joka sisältää tiedot tarvittavista muutoksista. [6] Näiden tiedostojen erilaisia esitysmuotoja käsitellään myöhemmin.

Muodollisesti määriteltynä diff-algoritmin toiminta voidaan esittää näin:

$$F_{perusversio} + F_{muutettu} \rightarrow \Delta_{(perusversio,muutettu)} \tag{2.1}$$

Muutostiedosto (Δ) saadaan siis antamalla algoritmille syötteeksi tiedoston F kaksi eri versiota, jolloin tulokseksi saadaan tiedosto, joka esittää näiden versioiden välillä tehdyt muutokset. Saadun muutostiedoston avulla voidaan puolestaan siirtyä molempien versioiden välillä, kunhan vähintään toinen niistä tunnetaan:

$$F_{perusversio} + \Delta_{(perusversio,muutettu)} \rightarrow F_{muutettu} \tag{2.2}$$

$$F_{muutettu} - \Delta_{(perusversio,muutettu)} \rightarrow F_{perusversio} \tag{2.3}$$

Edellisessä tapauksessa delta on ns. tavallinen delta, eli se sisältää perusversion ja muutetun version erot. Tällöin perusversiosta voidaan tehdä muutettu versio lisäämällä siihen deltan sisältämät muutokset.

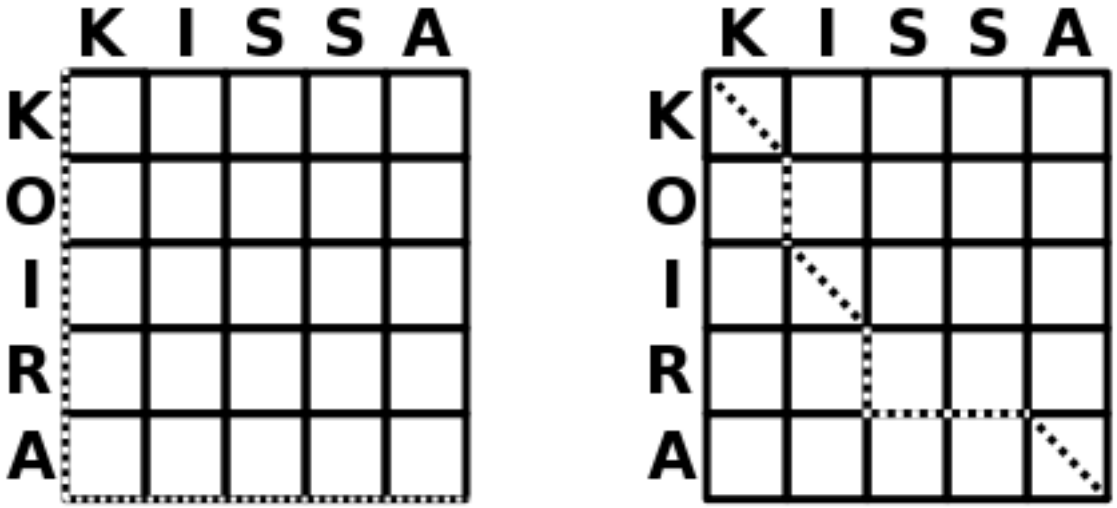
Jälkimmäisessä tapauksessa deltaa käytetään toisinpäin, eli sen avulla muutetusta versiosta tehdään perusversio poistamalla siitä deltan sisältämät muutokset. Tätä kutsutaan deltan *käänteiseksi soveltamiseksi* (engl. reverse apply).

Seuraava esimerkki osoittaa diff-algoritmin periaatteellisen toiminnan. Esimerkissä etsitään merkkijonojen “KISSA” ja “KOIRA” erot. Käytännössä eroavuuksia ei etsitä yksittäisten merkkien tarkkuudella, joten esimerkissä jokaisen kirjaimen voidaan ajatella vastaavan kokonaista tekstiriviä.

Vaihe 1: merkkijonot laitetaan taulukkoon.

Molemmat merkkijonot laitetaan taulukkoon kuvan 2.3 (vasen puoli) osoittamalla tavalla. Alkuperäinen merkkijono laitetaan pystysuoraan ja uusi merkkijono vaakasuoraan.

Vaihe 2: etsitään reitti yläkulmasta alakulmaan



Kuva 2.3: Diff-algoritmin toiminta.

Taulukossa (ks. kuva 2.3, vasen puoli) on tavoitteena siirtyä vasemmasta yläkulmasta oikeaan alakulmaan siten, että jokaisella askeleella joko lisätään tai poistetaan jokin kirjain. Jos molemmissa merkkijonoissa on sama kirjain, se voidaan ottaa lopputulokseen sellaisenaan.

Esimerkissä siirtymät tehdään siten, että oikealle siirryttäessä lopputulokseen lisätään vaakasuoran merkkijonon kirjaimia, ja alaspäin siirryttäessä poistetaan pystysuoran merkkijonon kirjaimia.

Helpoin tapa päästä alareunaan on siirtyä ensin suoraan alas ja sen jälkeen oikealle. Kuvan 2.3 vasemmanpuoleisessa taulukossa kuljettu reitti on merkitty katkoviivalla.

Taulukosta voidaan lukea suoraan seuraavat muutokset:

1. poistetaan k -k
2. poistetaan o -o
3. poistetaan i -i
4. poistetaan r -r
5. poistetaan a -a
6. lisätään k +k
7. lisätään i +i
8. lisätään s +s
9. lisätään s +s
10. lisätään a +a

Lopputuloksena syntyvä muutostiedosto on kutakuinkin tämän näköinen:

$-K, -O, -I, -R, -A, +K, +I, +S, +S, +A$

Vaikka lopputulos on toimiva, se ei ole optimaalinen. Tässä muodossa muutos tarkoittaisi koko alkuperäisen merkkijonon poistamista ja uuden merkkijonon lisäämistä. Parempi tapa on käyttää hyödyksi tilanteita, joissa merkkijonoilla on yhteisiä kirjaimia.

Parannellun algoritmin ensimmäinen vaihe eli merkkijonojen laittaminen taulukkoon pysyy lähes entisellään. Lisäyksenä entiseen vaiheeseen on se, että taulukkoon merkitään nyt ne kohdat, joissa molemmissa merkkijonoissa on sama kirjain. Merkintä on esitetty kuvan 2.3 oikeanpuoleisessa taulukossa.

Vaihe 2 eli reitinetsintä poikkeaa aikaisemmasta nyt siten, että suoraviivaisen etenemisen sijasta voidaan käyttää edellisessä vaiheessa lisättyjä "oikopolkuja". Käytännössä tämä tapahtuu käyttämällä jotain lyhimmän polun löytävää algoritmia.

Kuvan oikeanpuoleiseen taulukkoon katkoviivalla merkittyä reittiä seuraamalla päädytään seuraavanlaisiin muutoksiin:

1. säilytetään k k
2. poistetaan o -o
3. säilytetään i i
4. poistetaan r -r
5. lisätään s +s
6. lisätään s +s
7. säilytetään a a

Voidaan huomata, että muutosten määrä on pienempi, kuin suoraviivaista reittiä käyttävässä tilanteessa. Syntynyt muutostiedosto näyttää nyt tältä:

$K, -O, I, -R, +S, +S, A$

Yhteisten merkkien huomioonotto vaikuttaa muutostiedoston kokoon positiivisesti.

Esimerkissä tarkasteltu tilanne oli varsin yksinkertainen ja saattoi tuntua jopa turhalta näin pienen datamäärän tapauksessa. Todellisessa tilanteessa jokainen kirjain voisi kuitenkin olla jopa useiden megatavujen kokoinen datalohko, jolloin jokainen merkin lisäys tai poisto veisi niin paljon tilaa, että niiden niiden välttäminen olisi aiheellista. Oikopolkuja käytettäessä koko datalohkoa ei tarvitsisi tallentaa, koska sen tilalla olisi vain merkintä siitä, että tämä lohko pysyy ennallaan.

2.11 Tunnisteet

Lähes kaikissa versionhallintaohjelmissa tehdyille muutoksille annetaan yksikäsitteiset tunnisteet, joilla niihin voidaan viitata. Näiden tunnisteiden laatu vaihtelee aina juoksevasta numerosta kryptografisesti vahvaan hajautusarvoon, joten niiden käyttömahdollisuudet poikkeavat toisistaan melkoisesti.

Yksikäsitteiset nimet ovat eri järjestelmissä melko erilaisia, kuten taulukosta 2.1 voi nähdä.

Ohjelma	**Tunniste**
Arch	`matti@meikalainen.fi--archive-2006/mainline--1.0--patch-3`
Bazaar	`matti@meikalainen.fi-20060927202832-9795d0528e311e31`
BitKeeper	`matti@meikalainen.fi|ChangeSet|20060927183805|57296`
BitKeeper	`42516681VmgTWL0bkLcltPGiI6Yk5Q`
GIT	`81d4af1340badcd2100c84fbd1bfd13156de41aa`

Taulukko 2.1: Esimerkkejä muutoksille annettavista tunnisteista.

Archin tunniste muodostuu käyttäjän ja käytettävän tietovaraston tiedoista sekä juoksevasta numerosta. BitKeeperin kaksi tunnistetta ovat ns. *ChangeSet Key* ja *MD5 Key*, joista ensimmäinen perustuu Bazaarin tavoin käyttäjätietoihin ja muutosaikaan, ja jälkimmäinen muutoksista laskettuun tarkistussummaan[6]. GITin tunniste puolestaan perustuu kryptografisen tiivistefunktion[7] tiedoston sisällöstä laskemaan arvoon.

2.12 Muutosten esittäminen

Erilaisten diff-algoritmien avulla saadaan selville tehdyt muutokset. Muutokset pitää kuitenkin esittää sopivassa muodossa, jotta niistä olisi hyötyä.

Muutosten esitystapa riippuu suurelta osin käyttötarkoituksesta. Jos esimerkiksi on löydettävä kahdesta piirroksesta muutama ero, helpoin tapa on ympyröidä eroavat kohdat. Tällöin katsoja voi helposti verrata merkittyjä kohtia ja nähdä nii-

[6] BitKeeperin käyttämien tarkistusten ansiosta yritys ujuttaa Linux-ytimeen takaportti jäikin vain yritykseksi, koska se huomattiin ajoissa.

[7] Tiivistefunktio tarkoittaa funktiota, joka laskee saamalleen parametrille ns. tiivisteen eli yksilöllisen sormenjäljen. Kryptografinen tiiviste on käytännössä yksilöllinen, mutta pelkästään sen avulla ei voi selvittää alkuperäistä parametriä.

den eroavuudet.

Sama tapa voi toimia silloin, kun tiedostoja on muutettu vain vähän. Heti kun muutokset kasvavat riittävän suuriksi, tämä ei enää kelpaa, vaan on löydettävä parempia keinoja.

Seuraavissa alaluvuissa käsitellään muutamia muutosten esittämiseen käytettyjä menetelmiä.

2.12.1 QED-skripti

QED oli Butler Lampsonin ja Peter Deutschin kehittämä tekstieditori[8]. Ken Thompson kirjoitti siitä uuden version MITin CTSS:lle (Compatible Time Sharing System), lisäten mm. tuen *säännöllisille lausekkeille* (engl. regular expressions). Dennis Richie toteutti ohjelman GE-TSS:lle. Tätä versiota käytettiin skriptikielenä esimerkiksi eräajotiedostojen lähettämiseen suoritettavaksi ja tulostettavien tiedostojen muotoiluun.

Myöhemmin Ken Thompson toteutti *ed*-editorin, joka on nykyisin Unix-järjestelmien mukana tuleva perusohjelma. Ed oli QED:tä huomattavasti yksinkertaisempi ja ominaisuuksiltaan vaatimattomampi.

QED-skripteillä tiedostoja pystyttiin muokkaamaan eri tavoin. Tältä kannalta katsottuna skriptit olivat eräänlaisia muutostiedostoja, sillä ne kuvasivat, miten muutos tehdään. Myöhemmin skriptien teko onnistui myös automaattisesti, kun esimerkiksi Unixin *diff*-ohjelma osasi esittää muutokset Ed-editorin skriptinä. Kun tällainen skripti ajettiin, se teki tarvittavat muutokset alkuperäiseen tiedostoon.

Taulukossa 2.2 esitetään Ed-editorin käyttämä muutosskripti, joka muuttaa vasemmanpuoleisen tekstitiedoston oikeanpuoleiseksi. Käytännössä skripti lisää rivin 4 jälkeen sanan “kana” ja poistaa sitten rivit 3 ja 1.

Ed-skripti	Alkuperäinen tiedosto	Muutettu tiedosto
4a kana . 3d 1d	1. kissa 2. koira 3. kana 4. lehmä	1. koira 2. lehmä 3. kana

Taulukko 2.2: Ed-muutosskripti

[8] http://www.cs.bell-labs.com/cm/cs/who/dmr/qed.html (viitattu 10.8.2007).

2.12.2 Context diff

Context diff on yksinkertainen muutostenesitystapa, jossa alkuperäinen ja muutettu kohta esitetään peräkkäin. Muutoskohdan ympäriltä näytetään jonkin verran muuttumatonta ympäristöä eli ns. kontekstia, jonka avulla muutoskohdan sijainti voidaan tunnistaa.

Tämä tapa muistuttaa luvun alussa esitettyä keinoa muutoskohtien ympyröinnistä. Ympyrä määrittää näytettävän kontekstin.

Seuraava esimerkki näyttää, kuinka yksinkertainen muutos esitetään context diff -muodossa.

```
*** /usr/BACKUPS/ml.el.~2~  2007-09-04 10:55:20.000000000 +0300
--- /home/arirop/ml.el  2007-09-04 10:56:17.000000000 +0300
***************
*** 3,7 ****
    (let ((url w3m-current-url))
      (switch-to-buffer nil)
      (let ((pos (point)))
!       (insert "\n" "\footnote{" url "}\n")
        (goto-char pos))))
--- 3,7 ----
    (let ((url w3m-current-url))
      (switch-to-buffer nil)
      (let ((pos (point)))
!       (insert "\n" "\\footnote{" url "}\n")
        (goto-char pos))))
```

Context diff -esitysmuodon kahdella ensimmäisellä rivillä ilmoitetaan vertailtavat tiedostot aikamerkintöineen. Esimerkkitapauksessa vertailtavana on kaksi `ml.el`-tiedoston eri versiota, joita on viimeksi muokattu syyskuun alussa 2007.

Varsinaisen muutoksen ympäriltä näytetään jonkin verran muuttumatonta ympäristöä eli kontekstia, joka auttaa paikallistamaan kyseisen kohdan. Varsinainen muutoskohta osoitetaan rivin alussa olevalla huutomerkillä.

Muutoskohta konteksteineen näytetään kahteen kertaan. Ylempänä esitetään tiedoston sisältö ennen muutosta ja alempana sen jälkeen. Vertaamalla näitä kohtia silmämääräisesti voidaan havaita, ennemmin tai myöhemmin, että esimerkin ainoa muutos on yhden "\"-merkin lisääminen. Ilmeisesti alkuperäisessä koodissa on ol-

lut virhe, sillä Emacs Lispissä lainausmerkkien sisällä jokainen kenoviiva on tuplattava, jottei sitä tulkittaisi erikoismerkiksi.

Koneellisesti tulkittuna tämä esitysmuoto toimii vallan hyvin, mutta silmämääräisesti tarkasteltuna jotkut muutokset voivat olla hyvinkin vaikeasti havaittavia. Kontekstin esittäminen kahteen kertaan vie myös paljon tilaa, jota varsinkin versionhallinnan alkuaikoina pyrittiin säästämään. Näitä ongelmia on myöhemmin pyritty ratkomaan kehittämällä muutoksille uusi esitysmuoto.

2.12.3 Unified diff

Unified context diff (lyhyemmin unified diff tai unidiff) on context diff -esitysmuodon paranneltu versio. Kun context diff esittää alkuperäisen kohdan ja muutetun kohdan peräkkäin, tämä Wayne Davisonin kehittämä esitysmuoto näyttää ne limittäin[9]. Tätä ihmiselle havainnollisempaa muotoa käytetään etenkin Linux-kehityksessä käytettävissä *patch-tiedostoissa*[10], jotka näyttävät jokseenkin seuraavilta [17]:

```
--- a/fs/binfmt_elf.c
+++ b/fs/binfmt_elf.c
@@ -1152,7 +1152,7 @@ static int dump_write(struct file *file,
 static int dump_seek(struct file *file, loff_t off)
 {
        if (file->f_op->llseek && file->f_op->llseek != no_llseek) {
-               if (file->f_op->llseek(file, off, 1) != off)
+               if (file->f_op->llseek(file, off, SEEK_CUR) < 0)
                        return 0;
        } else {
               char *buf = (char *)get_zeroed_page(GFP_KERNEL);
```

Tässä muutos esitetään vanhan rivin poistamisena ja uuden lisäämisenä, joten sen ymmärtäminen on helppoa: tästä voi selvästi nähdä, että vakio 1 on korvattu symbolilla SEEK_CUR ja paluuarvon tarkistus on korjattu. Koska tarkasteltavat rivit ovat allekkain, niiden esittämät muutokset on helppo havaita myös silmämääräisesti.

[9] Unified diff -muotoa ei ole vielä standardoitu [34], mutta asia on muuttumassa: katso esim. http://www.opengroup.org/austin/mailarchives/ag-review/msg02077.html (viitattu 9.11.2006).

[10] Yleisiä suomenkielisiä nimityksiä ovat mm. *muutostiedosto* ja *päivitys*, mutta joskus puhutaan jopa *paikkauksista* tai puolikielisesti *pätseistä*.

2.12.4 Combined diff

Perinteisesti unified diff -muotoa on käytetty silloin, kun on haluttu selvittää kahden eri tiedoston väliset eroavuudet. Joskus on kuitenkin niin, että yhtä tiedostoa on kehitetty useassa eri *kehityshaarassa* (engl. branch) ja nämä haarat halutaan yhdistää. GITissä tämän helpottamiseksi on kehitetty *combined diff* -muoto, joka näyttää tältä:

```
diff --combined describe.c
@@@ +98,7 @@
    return (a_date > b_date) ? -1 : (a_date == b_date) ? 0 : 1;
   }

- static void describe(char *arg)
  -static void describe(struct commit *cmit, int last_one)
++static void describe(char *arg, int last_one)
   {
  +     unsigned char sha1[20];
  +     struct commit *cmit;
        struct commit_list *list;
        static int initialized = 0;
        struct commit_name *n;
```

Tässä tapauksessa tiedostolla describe.c on ollut kaksi vanhempaa, jotka on yhdistetty uudeksi versioksi. Uudessa versiossa kaksi ensimmäistä riviä esiintyivät ensimmäisessä muttei toisessa versiossa; kolme alinta riviä olivat mukana jo molemmissa aiemmissa versioissa.

2.12.5 Muutosten ryhmittely

Monissa tapauksissa yhtä loogista muutosta varten on muutettava useaa eri tiedostoa. Tällainen tilanne tulee eteen esimerkiksi silloin, kun monista paikoista kutsuttavan kirjastofunktion parametrejä muutetaan, jolloin sovellus ei toimi, ennen kuin kaikki kutsupaikat on päivitetty.

Jos käytettävä versionhallintajärjestelmä käsittelee kokonaisuutta yksittäisinä tiedostoina, yksinkertainen edellä kuvatun kaltainen muutos voidaan joutua kirjaamaan useana erillisenä muutoksena, joista jokainen muuttaa vain yhtä tiedostoa. Tässä on kuitenkin se ongelma, että näin kehityshistoriaan tulee kohtia, joiden aikana sovellus ei toimi.

Koska looginen muutos toimii oikein vain yhtenä kokonaisuutena, sen pilkkominen osiin ei ole järkevää. Tämän vuoksi käyttöön on otettu ns. *atomiset muutokset*, joissa muutokset voidaan pitää loogisina kokonaisuuksina. Näitä kokonaisuuksia nimitetään eri sovelluksissa eri nimillä, mutta suomeksi voidaan puhua kuvaavasti *muutosjoukoista* (engl. ChangeSet, ChangeGroup, Commit...). Collins-Sussman ja kumppanit toteavat tämän käsitteen sisällön vaihtelevan jonkin verran, mutta päätyvät antamaan seuraavan määritelmän:[8]

> For our purposes, let's say that a changeset is just a collection of changes with a unique name. The changes might include textual edits to file contents, modifications to tree structure, or tweaks to metadata. In more common speak, a changeset is just a patch with a name you can refer to.

Annettu määritelmä on sekä käytännönläheinen että täsmällinen. Vaikka sitä ei suoraan sanotakaan, muutosjoukkojen tarkoituksena on pitää loogiset muutokset yhtenäisinä kokonaisuuksina niin, että jokainen joukko sisältää vain toisiinsa kiinteästi liittyviä muutoksia.

2.13 Muutosten yhdistäminen

Shen [61] määrittelee *muutosten yhdistämisen* (engl. merging) seuraavasti:

> A critical function for SCM systems to support concurrent software development is merging, which is the process of integrating different pieces of work concurrently done by different developers on the same source code.

Buffenbarger ja Gruell [5] määrittelevät käsitteen samalla tavalla:

> [Merge:] The process of creating a new version of an element from a predecessor version and at least one other contributor version.

Yhdistäminen (engl. merging) tarkoittaa toimintaa, jossa saman tiedoston kahdesta tai useammasta versiosta muodostetaan uusi versio, johon otetaan mukaan muutoksia eri versioista. Lopputuloksena syntyvä tiedosto on siis eräänlainen yhdiste.

Tavallisesti usean tiedoston yhdistäminen tapahtuu yhdistämällä ensin kaksi tiedostoa, ja käyttämällä näin saatua uutta tiedostoa seuraavan yhdistyksen toisena osapuolena. Kahden tiedoston yhdistämiseen on olemassa kolme erilaista tapaa:

- Jos molemmat aikaisemmat versiot ovat sisällöltään samat, riittää, että uudeksi versioksi valitaan suoraan jompi kumpi.
- Jos toinen aikaisempi versio on selvästi toista parempi, on mahdollista, että huonompi versio jätetään kokonaan pois, ja uudeksi versioksi valitaan parempi versio sellaisenaan.
- Yleisin tilanne on, että uusi versio kootaan kahden aikaisemman version pohjalta siten, että molemmista otetaan mukaan parhaat osat, ja siten saatua tulosta muokataan jonkin verran.

Näiden lisäksi on tietenkin mahdollista, että molemmat aikaisemmat versiot ovat niin huonoja, että uusi versio kannattaa aloittaa kokonaan tyhjästä – tällöin kyseessä ei kuitenkaan ole tiedoston uusi versio vaan kokonaan uusi tiedosto.

Kuten edeltä käy ilmi, yhdistämiseen vaaditaan vähintään kahden eri version olemassaoloa, mikä onkin helppo ymmärtää: uuden tiedoston luonti ei ole yhdistämistä, eikä yhdessä versiossa voi olla eroavaisuuksia itsensä suhteen. Tämän havainnon perusteella versionhallinta voidaankin jakaa kahteen luokkaan, pessimistiseen ja optimistiseen versionhallintaan. *Pessimistisessä* tapauksessa käsiteltävät tiedostot lukitaan siten, että vain yksi käyttäjä kerrallaan voi muokata niitä; samanaikaista kehitystä ei siis ole, joten yhdistämiselle ei ole tarvetta. *Optimistisessa* tapauksessa kukin käyttäjä muokkaa omaa kopiotaan tiedostosta, josta seuraa se, että jossakin vaihteessa tehdyt muutokset on yhdistettävä yhteiseen versioon. [44]

Mikäli varsinaista yhdistämistä tarvitaan, siihen on olemassa useita erilaisia menetelmiä; useimmissa tapauksissa menetelmä kuitenkin vain ilmoittaa, missä on ristiriita ja jättää sen ratkaisemisen käyttäjän vastuulle.

2.13.1 Yhdistämismenetelmien jaottelu

Yhdistämismenetelmiä voidaan luokitella niiden toiminnan mukaan varsin monella eri tavalla. Karkea jako kahteen voidaan tehdä sen perusteella, miten ristiriidat alun perin havaitaan: tällöin *tilapohjaiseen* (engl. state based) luokkaan kuuluvat ne menetelmät, joissa analysoidaan versioiden välisiä eroavuuksia, ja *muutospohjaiseen* (engl. change based) luokkaan ne, joissa ristiriidat havaitaan tehtyjä muutoksia tutkimalla. [44, 71]

Muita luokitteluja lähteen [61] mukaan voisivat olla esimerkiksi tekstipohjainen vs. oliopohjainen, syntaktinen vs. semanttinen sekä aiemmin mainittu jaottelu tila-

pohjaisen ja toimintopohjaisen (muutospohjaisen) välillä. Luokittelu on varsin kirjavaa, joten joissakin tapauksissa toimintopohjaisia menetelmiä pidetään muutospohjaisten menetelmien alaluokkana, mutta toisissa niitä pidetään synonyymeinä; tässä tutkielmassa nämä menetelmät pidetään erillään.

Tilapohjaisissa menetelmissä analysoidaan yleensä eri versioiden välisiä eroavuuksia esimerkiksi diff-algoritmin avulla.

Muutospohjaisissa menetelmissä tehdyt muokkaukset esitetään yleensä deltoina, joita seuraamalla voidaan selvittää eri versioiden kehityshistoriat. Näihin menetelmiin kuuluvissa *toimintopohjaisissa menetelmissä* (engl. operation based) tietoja ei voikaan muokata aivan vapaasti, vaan muutokset tehdään aina jonkinlaisina tarkasti määrättyinä operaatioina. [44]

Tila- ja muutospohjaisten menetelmien ero on käytännössä siinä, että tilapohjaisessa tehdyt muutokset pitää selvittää annettujen versioiden perusteella, mutta muutospohjaisessa tehdyistä muokkauksista on pidetty koko ajan lokia. Ensimmäisessä tapauksessa ostetut tavarat pitäisi siis päätellä kahden valokuvan perusteella (jääkaapin sisältö ennen ja jälkeen kauppareissun) ja toisessa päätelmät voisi tehdä ostoskuittia tutkimalla.

Eroja näiden menetelmien välillä alkaa näkyä silloin, jos alku- ja lopputilanteen välillä tehdään useita muutoksia. Jos esimerkiksi ensin muutetaan kaikki *A*-kirjaimet *B*-kirjaimiksi ja sitten *B*-kirjaimet *C*-kirjaimiksi, niin tilapohjaisen menetelmän kannalta ainoa tehty muutos on siinä, että kaikki *A*:t on muutettu *C*:iksi. Tämä ongelma kuitenkin ratkeaa helposti niin, että uusi tila luodaan sekä ensimmäisen että toisen muutoksen jälkeen, jolloin muutokset voidaan päätellä kahdessa vaiheessa. Muutospohjaisissa menetelmissä ongelmaa ei tule, koska kaikki muutokset kirjataan ylös jo niitä tehdessä.

Edellä esitettyjen lisäksi menetelmiä voidaan luokitella sen mukaan, millä tasolla niissä käsitellään tietoja [44]:

Tekstipohjaisessa yhdistämisessä (engl. textual merging) tietoa käsitellään puhtaana tekstinä, joten ristiriidat ovat samaan kohtaan tehtyjä päällekkäisiä muokkauksia. Tämä on yleisin ja varsin luotettava menetelmä, mutta vaatii käyttäjän osallistumista.

Syntaktisessa yhdistämisessä (engl. syntactic merging) tietoa käsitellään ohjelmointikielen tasolla, joten toteutus on sidottu aina tiettyyn kieleen. Menetelmä havaitsee ne ristiriidat, jotka aiheuttavat syntaksivirheitä, mutta epäoleelliset (esimerkiksi kommentteihin tehdyt) muutokset jäävät huomiotta.

Semanttisessa yhdistämisessä (engl. semantic merging) käsitellään ohjelmia niiden toiminnallisuuden perusteella, joten ristiriitoja aiheuttavat toiminnan muutokset havaitaan. Menetelmä osaa esimerkiksi varoittaa, jos kulmat lasketaan muutoksen jälkeen radiaaneina asteiden sijaan.

Rakenteellisessa yhdistämisessä (engl. structural merging) ohjelma pitää päivittää esimerkiksi luokkahierarkiassa tapahtuneiden muutosten vuoksi. Muutosten aiheuttamia mahdollisia ongelmia ei ole aina mahdollista havaita ohjelmallisesti, joten tässä vaaditaan yleensä käyttäjän aktiivista osallistumista.

Edellä mainitut ristiriidat tai konfliktit ovat tilanteita, joissa muutoksia ei voi yhdistää automaattisesti. Nämä tilanteet voidaan luokitella oikeiksi tai virheellisiksi sen mukaan, mistä ne aiheutuvat.

Oikea ristiriita on tilanne, jossa muutoksia ei voi yhdistää oikein: esimerkiksi uuden version päivämäärää ei voi ottaa jommasta kummasta aikaisemmasta versiosta, vaan se riippuu muokkausajankohdasta. Virheellinen ristiriita on tilanne, jossa ei oikeasti ole ristiriitaa, mutta käytettävä menetelmä luulee havaitsevansa sellaisen. Esimerkkinä virheellisestä ristiriidasta voidaan mainita tilanne, jossa kumpikin versio sisältää saman muutoksen. Tällöin muutokset törmäävät keskenään, mutta eivät ole ristiriitaisia, sillä ne ovat identtiset[11].

Täysin automaattinen yhdistäminen on hyvin vaikeaa ja yleensä mahdollista vain tarkasti rajoitetuissa tilanteissa. [44]

2.13.2 2-way-merge ja 3-way-merge

2-way-merge ja *3-way-merge* ovat yhdistämistapoja, joissa yhteinen versio muodostetaan kahden tai kolmen aikaisemman version perusteella: kahden version tapauksessa molemmat versiot ovat toisistaan riippumattomia, mutta kolmen tapauksessa yhden version on oltava kahden muun yhteinen *esi-isä* (engl. parent). [44]

Molemmat menetelmät ovat tilapohjaisia, joten niissä tarkastellaan ainoastaan tiedostojen välisiä eroavuuksia. Tähän tarkoitukseen käytetään tavallisimmin jotakin aiemmin mainitun diff-algoritmin muunnosta.

Seuraavat esimerkit havainnollistavat näiden menetelmien eroa.

[11] Identtiset muutokset voivat tosin aiheuttaa semanttisia konflikteja. Jos esimerkiksi jokin vakio sisältää käytettävien alkioiden määrän, ja kahden kehittäjän muutokset kasvattavat sitä yhdellä, lopputulos ei ole odotettu: esimerkiksi molempien tekemä muutos $A = 123 \longrightarrow A = 124$ ei ole oikein, sillä lopputuloksen pitäisi olla $A = 125$.

esi-isä	variantti 1	variantti 2	sääntö	lopputulos
kana	kana	kana	kana = kana → kana	kana
kana	emu	kana	emu ≠ kana → konflikti	konflikti
kana	kana	käki	kana ≠ käki → konflikti	konflikti
kana	emu	emu	emu = emu → emu	emu
kana	emu	käki	emu ≠ käki → konflikti	konflikti

Taulukko 2.3: Esimerkki 2-way-mergen säännöistä.

Taulukossa 2.3 kuvataan 2-way-mergen toimintaa tilanteessa, jossa alkuperäistä tiedostoa on muutettu kahdessa eri kehityshaarassa, jolloin siitä on muodostunut kaksi varianttia. Kuten esimerkeistä voi huomata, tässä menetelmässä esi-isä jätetään täysin huomiotta, jolloin lopputulokseen vaikuttaa vain muutettujen tiedostojen sisältö.

esi-isä	variantti 1	variantti 2	sääntö	lopputulos
kana	kana	kana	kana = kana = kana → kana	kana
kana	emu	kana	kana = kana ≠ emu → emu	emu
kana	kana	käki	kana = kana ≠ käki → käki	käki
kana	emu	emu	kana ≠ emu = emu → emu	emu
kana	emu	käki	kana ≠ emu ≠ käki → konflikti	konflikti

Taulukko 2.4: Esimerkki 3-way-mergen säännöistä.

Taulukossa 2.4 esitetään sama tilanne 3-way-mergen kannalta. Tilanne poikkeaa aiemmasta siinä, että yhdistämisessä otetaan huomioon myös esi-isä, jolloin konfliktien määrä vähenee selvästi.

Kuten esimerkit osoittavat, 3-way-merge on näistä kahdesta luotettavampi ja kykenee tunnistamaan useampia virheellisiä konflikteja, eli tilanteita, joissa ristiriitaa ei oikeasti ole. Sen toiminta perustuu seuraavalle yksinkertaiselle algoritmille [15]:

- Etsitään yhteinen esi-isä, johon muutoksia verrataan.
- Jos muutos on tehty vain toisessa kehityshaarassa, se otetaan mukaan sellaisenaan.
- Jos molemmissa kehityshaaroissa on tehty sama muutos, se otetaan mukaan.
- Muussa tapauksessa ristiriidasta ilmoitetaan käyttäjälle, jonka tehtäväksi jää sen ratkaiseminen.

Yksinkertaisuutensa ja luotettavuutensa vuoksi 3-way-merge onkin vakiinnuttanut paikkansa monien sovellusten vakioyhdistysmenetelmänä. Aivan kaikkiin tilanteisiin se ei kuitenkaan sellaisenaan sovi, mutta näitä ongelmatilanteita ja niihin esitettyjä ratkaisuja käsitellään myöhemmin eri sovellusten yhteydessä.

2.13.3 Konfliktien välttäminen

Monen kehittäjän projekteissa konflikteja ei voi aina välttää, mutta niiden määrää voidaan yrittää vähentää. Mens [44] esittää yksinkertaiseksi keinoksi sen, että ohjelmoijat pidetään ajan tasalla tehtävistä muutoksista. Kun kaikki tietävät missä mennään, kukin osaa välttää päällekkäisten muutosten tekoa, tai ainakin tuleviin ristiriitoihin osataan varautua.

Mens mainitsee myös Brueggen ja Dutoit'n kaksi heuristiikkaa:

- Pääkehityshaaraan tehdään vain bugikorjauksia ja muut muutokset tehdään erillisiin kehityshaaroihin.
- Kehityshaarojen määrä pidetään mahdollisimman pienenä.

Vaikka hajautetussa kehityksessä – kuten Linux-ytimen tapauksessa – käytettävien kehityshaarojen määrä onkin edellisten heuristiikkojen vastaisesti suuri ja päällekkäiset muutokset jokapäiväisiä, niistä ei käytännössä aiheudu juurikaan ongelmia. Yksinkertaiseen 3-way-mergeen pohjautuvat algoritmit selvittävät suurimman osan yksinkertaisista tapauksista luotettavasti, ja muissa tapauksissa tarvitaan joka tapauksessa kehittäjän osallistumista. Näin ollen kovin paljon älykkäämmille yhdistysmenetelmille ei ole ainakaan Linux-kehityksen yhteydessä vielä mitään todellista tarvetta. [24]

2.14 Uudelleennimeäminen

Ohjelmistoja kehitettäessä lähdekooditiedostot pyritään nimeämään niiden sisältöä kuvaavasti. Kun kehitysprojekti etenee, koodin lisääntyminen ja muuttuminen saattaa johtaa siihen, että tiedostojen sisältö ei enää vastaa niiden nimeä. Isommissa projekteissa puolestaan tiedostojen määrä saattaa kasvaa niin suureksi, että niitä pitää järjestellä uudelleen järjestyksen säilyttämiseksi.

Uudelleennimeämisellä (engl. renaming) tarkoitetaan normaalisti sitä, että jonkin objektin, kuten tiedoston tai hakemiston, nimi vaihdetaan toiseksi. Versionhallinnan

yhteydessä tilanne on käytännössä sama, mutta siinä pitää ottaa huomioon myös nimettävän objektin kehityshistoria. On tärkeää, että vanhaan nimeen liitetty historia ja muissa kehityshaaroissa tehdyt muutokset osataan myöhemmin yhdistää uudella nimellä olevaan tiedostoon.

Uudelleennimeämiseksi katsotaan myös se, että tiedosto siirretään toiseen hakemistoon, koska tällöin tiedoston *absoluuttinen nimi* vaihtuu. Absoluuttisella nimellä tarkoitetaan tässä tiedoston sijaintia työhakemiston päähakemistosta katsoen, eikä tiedostojärjestelmän juuresta katsoen. Esimerkiksi tiedosto `a.c` voidaan siirtää alihakemistoon `src`, jolloin tiedoston absoluuttinen nimi vaihtuu `a.c`:stä muotoon `src/a.c`.

Versionhallintaohjelmat voivat käsitellä uudelleennimeämisiä joko *eksplisiittisesti* tai *implisiittisesti*. Edellisessä tapauksessa käyttäjän tehtävänä on ilmoittaa mahdollisista nimien muutoksista sovellukselle, kun taas jälkimmäisessä ne jäävät versionhallintaohjelman pääteltäviksi. Molemmilla tavoilla on omat etunsa ja omat haittansa.

Eksplisiittisessä tapauksessa pidetään erikseen yllä tietoja nimien vaihdoista, jolloin ne ovat oikein tehtyinä luotettavia. Ongelmia tulee kuitenkin silloin, jos nimiä vaihdetaan versionhallintaohjelman tietämättä. "Väärin" tehdyt toiminnot voivat vahingoittaa tietovarastoa esimerkiksi niin, ettei automaattinen yhdistäminen enää toimi.

Implisiittisessä tapauksessa ei ole mitään väliä sillä, miten tiedostoja nimetään. Eräs tapa säilyttää tiedoston identiteetti uudelleennimeämisestä huolimatta on liittää siihen samantapainen tunniste kuin aiemmin kohdassa 2.11 on esitetty. Esimerkiksi GNU Emacs -editorin mukana tulevan `vc-arch.el`-tiedoston lopussa on seuraavanlainen Arch-versionhallintaohjelman tunnisterivi eli *tagi*, jonka avulla tiedosto tunnistetaan myös nimenvaihdon jälkeen:

```
;; arch-tag: a35c7c1c-5237-429d-88ef-3d718fd2e704
```

Tällaiset tiedostokohtaiset tunnisteet aiheuttavat kuitenkin jonkin verran ongelmia, joten niiden käyttöä pyritään nykyään välttämään. Yksi ongelma on jo siinä, että tunniste koskee kokonaista tiedostoa. Jos tällainen tiedosto joskus pilkotaan vaikkapa kahteen osaan, niin jomman kumman puoliskon kehityshistoria häviää, koska molemmilla ei voi olla samaa tunnistetta[12].

Uudemmissa järjestelmissä tiedostotunnisteita ei enää juurikaan käytetä, vaan ne on korvattu esimerkiksi tiedoston sisällöstä laskettavalla tunnistearvolla. Koska

[12] http://www.gelato.unsw.edu.au/archives/git/0610/29818.html (viitattu 25.1.2007).

tiedoston sisältö ei nimenvaihdon yhteydessä muutu, sen identiteetti säilyy myös uudella nimellä, joten kehityshistoria osataan liittää oikeaan tiedostoon.

2.15 2000-luvun versionhallintaohjelmat

2000-luvun alussa oltiin aivan eri tilanteessa 1970-lukuun verrattuna: tavalliset ihmiset käyttivät tietokoneita, joiden tehokkuudet ylittivät vuosikymmenten takaisten supertietokoneiden suorituskyvyn, ja käytännöllisesti katsoen kaikki olivat yhteydessä toisiinsa Internetin välityksellä. Yksi asia ei kuitenkaan ollut suuremmin muuttunut – versionhallinta. Ihme kyllä, suuri osa silloisista järjestelmistä käytti edelleen vanhoja, joko SCCS:ään (esim. BitKeeper) tai RCS:ään (esim. Perforce) perustuvia ratkaisuja. Myös CVS:n seuraajaksi kaavailtu Subversion [8] oli jatkanut vanhalla linjalla korjaten vain joitakin CVS:n puutteita.

Eräs syy vanhojen tekniikoiden käyttöön oli tietenkin siinä, että ne toimivat hyvin, joten niitä ei tarvinnut suuremmin parannella. Ne eivät kuitenkaan kelvanneet aivan kaikille. Eräs esimerkki tästä on paljon seurattu Linux-ytimen kehitysprojekti: siinä ei itse asiassa käytetty lainkaan versionhallintaa, koska Linus Torvaldsin mielestä yksikään senhetkinen versionhallintaohjelma ei täyttänyt asetettuja vaatimuksia [3][13].

2.15.1 BitKeeper-kriisi

Linux on tullut tunnetuksi maailmanlaajuisena projektina, johon ottavat osaa tuhannet vapaaehtoiset ympäri maailmaa. Ongelmana oli, ettei missään järjestelmässä oltu osattu varautua näin massiivisesti hajautettuun kehitysmenetelmään – ohjelmat eivät yksinkertaisesti skaalautuneet riittävästi.

Jo vuonna 1998 oli käynyt selväksi se asia, ettei mikään versionhallintaohjelma toiminut Linux-kehityksessä. Kun Linus[14] alkoi osoittaa loppuunpalamisen merkkejä, huomattiin ettei tilanne voinut enää jatkua entisellään. Tämä yhdistettynä sil-

[13] On muistettava, että vaikka jotkut yksittäiset kehittäjät käyttivät versionhallintaa, se ei käytännössä merkinnyt mitään: muutokset oli joka tapauksessa lähetettävä sähköpostilla patch-tiedostoina.

[14] Tässä tutkielmassa noudatetaan Linus Torvaldsin tapaa, josta hän kertoo sähköpostiviestissään Junio C Hamanolle (junkio@cox.net) näin:

> ... I'm not "Torvalds", I'm "Linus". And like it or not, you're either Junio or junkio at least to me ;)

(http://www.gelato.unsw.edu.au/archives/git/0505/4007.html (viitattu 25.8.2007).)

loiseen kehitysmalliin johti siihen, että Larry McVoy tarjosi apuaan tilanteen parantamiseksi[15]. Helmikuussa 1999 hän ilmoitti kernel-kehitykseen tarkoitetusta uudesta versionhallintaohjelmasta, BitKeeperistä[16]:

> For those of you who don't know, BitKeeper is an Open Source distributed revision control system which I claim is a substantial step forward from CVS.

Mielenkiintoista tässä ilmoituksessa oli se, että BitKeeper mainitaan avoimen lähdekoodin ohjelmaksi, mikä ei lopulta pitänytkään paikkaansa.

Linus otti BitKeeperin käyttöönsä vuonna 2002 [60]. Ei-vapaan ohjelman käyttöönotto vaikutti syvästi vapaiden ohjelmien puolustajiin: nyt oltiin tilanteessa, jossa vapaiden ohjelmien lippulaivan, Linux-ytimen, kehityksessä käytettiin suljettua järjestelmää. Pelättiin, että tämä antaisi väärän signaalin suurelle yleisölle – jotakin oli tehtävä.

BitKeeperin lisenssi johti moniin keskusteluihin. Yhdeksi aiheeksi nousi mm. se seikka, että BitKeeperin ilmaista versiota ei saanut käyttää, mikäli kehitti jotain muuta versionhallintaohjelmaa. Richard Stallman aloitti aiheesta keskustelun, missä toivoi kiivaan lisenssikeskustelun johtavan BitKeeperin käytön lopettamiseen Linux-kehityksessä[17].

Vastaukset Stallmanin viestiin eivät kuitenkaan olleet kovin rakentavia. Joidenkin mielestä BitKeeperin käyttö oli hyvä asia, sillä parempaakaan vaihtoehtoa ei ollut tarjolla. Eräät vastaajat kritisoivat Stallmanin puuttumista asiaan, joka ei hänelle kuulunut. Larry McVoy tarttui tilaisuuteen kritisoida GPL:ää, kääntäen keskustelun hetkeksi muualle[18]. Tämä ja monet muut viestiketjut toivat selvästi esille eron vapaiden ohjelmien ja avoimen lähdekoodin kannattajien välillä.

Moni valitti BitKeeperin käytöstä, mutta vain harvat tekivät mitään konkreettista tilanteen korjaamiseksi.

Maaliskuussa 2002 kernel-listalle lähetettiin vetoomus, ettei BitKeeperin käyttöä suositeltaisi. Vetoomuksessa suositeltiin odottamaan Archin tai Subversionin valmistumista[19].

[15] Katso http://lkml.org/lkml/1998/9/30/122 (viitattu 8.11.2006).
[16] http://lkml.org/lkml/1999/2/21/15 (viitattu 8.10.2007).
[17] http://lkml.org/lkml/2002/10/13/201 (viitattu 8.10.2007).
[18] http://lkml.org/lkml/2002/10/19/163 (viitattu 8.10.2007).
[19] http://lkml.org/lkml/2002/3/5/204 (viitattu 8.10.2007).

Andrew Morton totesi kannattavansa vetoomuksessa esitettyjä perusteluja. Suurin osa muista vastaajista oli sitä mieltä, että vetoomuksia parempi tapa tilanteen parantamiseksi olisi paremman versionhallintaohjelman kirjoittaminen.

Tammikuussa 2003 Jamie Lokier ilmoitti ongelmista, joita BitKeeperin lisenssi aiheuttaa hänelle. Samalla hän kysyi, voisiko jostain saada BitKeeperin verkkoprotokollan dokumentaation. Larry McVoy vastasi, että kyseessä oleva protokolla on niin arvokas, ettei sitä tulla julkaisemaan[20].

Helmikuussa 2003 Pavel Machek ilmoitti tehneensä yksinkertaisen BitKeeperin tietoja lukemaan pystyvän BitKeeper-kloonin, BitBucketin[21]. Ben Collins julkisti puolestaan oman skriptinsä, jolla pystyi saamaan muutokset esille BitKeeper-tietovarastosta[22]. McVoy muistutti, että BitKeeper on tuotemerkki, joten BitBucketia ei saanut mainostaa sen kloonina. Tämä johti siihen, että muutamissa viesteissä puhuttiin kuvitteellisesta KitBeeper-ohjelmasta.

Heinäkuussa 2003 Larry McVoy uhkasi muuttaa BitKeeperin verkkoprotokollaa, jotta kilpailevat ohjelmat eivät toimisi sen kanssa. Hän oli valmis käyttämään digitaalisia allekirjoituksia ohjelmien yhteistoiminnan estämiseksi[23]. Richard Stallman tarttui tähän ilmoitukseen, todeten ajan olevan kypsä vapaan BitKeeper-asiakasohjelman tekemiseen. Mikäli McVoy toteuttaisi uhkauksensa, hän menettäisi Linux-kehittäjien tuen[24]. McVoy totesi tähän kilpailevan ohjelman tekemisen vaativan BitKeeperin toiminnan selvittämisen, mikä olisi mahdotonta ilman lisenssiehtojen rikkomista[25].

Tilanne kärjistyi lopulta vuonna 2005, kun *Samba*n ja *rsync*in kehittäjänä tunnetuksi tullut Andrew Tridgell sai analysoitua BitKeeperin käyttämän protokollan [56], ja toteutti sitä tukevan avoimen lähdekoodin *SourcePuller*-asiakasohjelman. McVoy totesi Tridgellin toiminnan rikkoneen lisenssiehtoja ja ilmoitti BitKeeperin ilmaisversion käyttölisenssien sen vuoksi raukeavan heinäkuun ensimmäisenä päivänä. Myös Linus arvosteli Tridgellin toimia kovin sanoin. Koska ilmaisen BitKeeperin käyttö ei voinut enää jatkua, Linux-kehittäjät joutuivat hankalaan tilanteeseen. Linusin ilmoitus BitKeeperin käytön lopettamisesta[26] aloitti uuden versionhallintaohjelman etsinnän.

[20] http://lkml.org/lkml/2003/1/18/5 (viitattu 8.10.2007).
[21] http://lkml.org/lkml/2003/2/26/248 (viitattu 8.10.2007).
[22] http://lkml.org/lkml/2003/3/1/173 (viitattu 8.10.2007).
[23] http://lkml.org/lkml/2003/7/17/124 (viitattu 8.10.2007).
[24] http://lkml.org/lkml/2003/7/18/260 (viitattu 8.10.2007).
[25] http://lkml.org/lkml/2003/7/18/281 (viitattu 8.10.2007).
[26] http://lkml.org/lkml/2005/4/6/121 (viitattu 8.11.2006).

Vaikka erilaisia järjestelmiä oli runsaasti saatavilla, mikään niistä ei ollut riittävä Linux-kehitykseen sopivaksi. Tämä saattoi johtua siitä, että Linuxin kehitys poikkesi huomattavasti perinteisistä ohjelmistokehitysprojekteista. Kukaan ei ollut osannut odottaa näin massiivisesti hajautetun kehitysprojektin syntyä.

2.15.2 GIT ja Mercurial

BitKeeperistä luopumisen jälkeen Linux-kehityksessä oltiin taas tilanteessa, jossa mitään versionhallintaa ei käytetty. Tämä tilanne ei kuitenkaan jatkunut kovin pitkään, sillä kokemuksista viisastunut Linus alkoi pian etsiä korvaavaa järjestelmää. Etsintöjen aikana hän tutustui mm. Monotoneen, mutta havaitsi sen riittämättömäksi. Hän sai siitä kuitenkin jonkin verran ideoita, joiden pohjalta hän alkoi kehittää omaa, aluksi väliaikaiseksi tarkoitettua järjestelmää; tuloksena syntyi GIT – ensimmäinen vapaa versionhallintaohjelma, joka soveltuu Linux-kehitykseen. [24]

Toinen samoihin aikoihin aloitettu versionhallintaohjelma on Pythonilla kirjoitettu Mercurial [43], joka suunniteltiin alusta alkaen mahdollisimman skaalautuvaksi[27]; sen olemassaolo osoittaa, että ns. BitKeeper-kriisin ansiosta alettiin vihdoin huomata, minkälaisia ohjelmia todella kaivattiin ja mitä ongelmia niiden on ratkaistava. Onkin suorastaan ihme, ettei näitä ongelmia oltu osattu ottaa vakavasti aiemmin!

Sekä GIT että Mercurial saivat vaikutteita useista aikaisemmista versionhallintaohjelmista, kuten kaupallisesta BitKeeperistä ja aiemmin mainitusta Monotonesta. Koska niiden ongelmakohdat tiedostettiin alusta alkaen, ne voitiin ottaa huomioon jo suunnitteluvaiheessa. Lopputuloksena syntyneiden ohjelmistojen voidaan katsoa aloittaneen jälleen uuden versionhallintaohjelmien sukupolven.

2.16 Versionhallinnan tulevaisuus

Nykyään versionhallinta on jo niin arkipäiväistä, ettei sen olemassaoloon kiinnitetä juurikaan huomiota. On varsin tavallista, että sovellukset antavat käyttäjän palata dokumenttien aikaisempiin versioihin ja tarkastella niihin tehtyjä muutoksia. Ongelmaksi tässä kuitenkin muodostuu se, että jokainen sovellus toteuttaa nämä ominaisuudet omalla tavallaan, joten historiaan pääsee käsiksi vain tietyllä sovelluksella. Ratkaisuksi tähän on esitetty mm. tiedostojärjestelmän tasolla toimivaa versioin-

[27] Katso http://lkml.org/lkml/2005/4/20/45 (viitattu 8.11.2006).

tia, joka tarjoaa yhteisen palvelun kaikille sovelluksille. [11]

Tulevaisuutta on tunnetusti vaikea ennustaa, ellei sitten ole itse tekemässä sitä. Vaikka kehitys ei aina menekään odotettuun suuntaan, jonkinlaisia perusteltuja arvauksia voi tehdä vertaamalla nykytilaa jo elettyyn historiaan ja katsomalla, kuinka uusia mahdollisuuksia on aiemmin hyödynnetty.

Verkkojen nopeutuminen mahdollistaa entistä suurempien tietomäärien siirron. Tietokoneiden nopeudet eivät ole enää kasvaneet niin rajusti kuin aiemmin, mutta ytimien määrä on moninkertaistunut. Näiden trendien pohjalta voidaan ennustaa, että tulevaisuudessa ohjelmistokehityksessä keskitytään entistä enemmän hajautukseen. Ohjelmista tehdään rinnakkaisia ja hyvin modulaarisia. On mahdollista, että ohjelmistot tulevat koostumaan hajautetuista moduuleista, joita ajetaan verkon yli. Tämä asettaa omat vaatimuksensa myös versionhallintaohjelmille, kun pienikin virhe voi rikkoa jopa miljoonien käyttäjien sovellukset.

Myös versioitava data monipuolistuu. Multimedian versiointi asettaa aivan uusia vaatimuksia, kun tekstipohjaiset algoritmit pitää päivittää uusia odotuksia vastaaviksi. Odotettavissa on myös tiedostojärjestelmien kehittyminen siten, että versionhallintatoiminnot sulautuvat niiden osaksi, jolloin versiointi muuttuu automaattiseksi. Tähän suuntaan oltiin jo menossakin mm. Hans Reiserin näkemysten mukaisessa Reiser4-tiedostojärjestelmässä[28], mutta nyt sen tulevaisuus näyttää epävarmalta. Ajan myötä kuitenkin selviää, oliko se itsessään huono idea, vai vain aikaansa edellä ollut väärinymmärretty edelläkävijä.

2.17 Yhteenveto

Versionhallinta on yksi konfiguraationhallinnan osa-alue. Lyhenteellä SCM voidaan tarkoittaa näistä kumpaa tahansa, mikä on omiaan aiheuttamaan sekaannusta. Tässä tutkielmassa ongelma pyritään kiertämään käyttämällä täsmällisempiä termejä.

Versionhallinnan käsitettä on määritelty eri aikoina eri tavoin, mutta käytännössä sillä tarkoitetaan versioituun informaatioon tehtävien muutosten hallintaa. Versiolla tarkoitetaan versioitavan alkion tilaa jonakin hetkenä, ja revisiolla sellaista versiota, joka korvaa edeltävän version. Variantti on puolestaan versio, joka on olemassa samanaikaisesti toisten versioiden kanssa.

Revisiot muodostavat lineaarisen kehityshistorian, ja variantit puolestaan puumaisen historian, jossa on kehityshaaroja. Näitä kehityshaaroja voidaan yhdistää

[28] http://lkml.org/lkml/2002/3/10/40/ (viitattu 24.10.2007).

erilaisten yhdistämismenetelmien avulla, jolloin puusta muodostuu graafi. Perinteisesti muokattavat versiot on suojattu samanaikaisilta muutoksilta lukitusten avulla, mutta niiden aiheuttamien ongelmien vuoksi niiden käyttöä pyritään nykyään minimoimaan.

Muutokset havaitaan yleensä jollakin diff-algoritmilla, ja ne voidaan esittää käyttötarkoituksesta riippuen usealla eri tavalla. Tavallisesti muutokset tallennetaan tietovarastoon tilaa säästävässä deltamuodossa, ja niihin lisätään jonkin verran metadataa, kuten tieto muutoksen ajankohdasta ja yksikäsitteinen tunniste, jolla muutokseen voidaan myöhemmin viitata.

Nykyään käytössä on perinteisten keskitettyjen versionhallintaohjelmien lisäksi myös hajautettuja vaihtoehtoja, jotka tuovat mukanaan sekä uusia mahdollisuuksia että uusia ongelmia.

3 Avoin lähdekoodi

> *We believe that the world of open source is both bigger and more varied that what literature describes, but we will anyhow present our findings by starting with a definition of open source. Open source is in our eyes three things, it is the software, licensed with an open source license; a set of development practices used to develop this software; and it is the community around this software.*
>
> — *Øyvind Hauge & Andreas Røsdal[26]*

Nyt kun versionhallinnan peruskäsitteet ovat hallinnassa, voidaan siirtyä avoimen lähdekoodin maailmaan. Tässä luvussa perehdytään tämän käsitteen merkitykseen ja historiaan sekä katsotaan, mitä uutta se on tuonut mukanaan. Esitys aloitetaan perinteiseen tapaan "aivan alusta" eli hakkerikulttuurin esittelyllä.

3.1 Hakkerikulttuuri

Termi "hakkeri" on nykyään usein esillä eri tiedotusvälineissä. Valitettavasti on enemmän sääntö kuin poikkeus, että sitä käytetään negatiivisessa merkityksessä. Richard Stallman, tunnettu hakkeri, on myös huomannut sanan väärinkäytön, mutta toteaa, ettei väärinkäytölle tarvitse antaa myöten[1]:

> The use of "hacker" to mean "security breaker" is a confusion on the part of the mass media. We hackers refuse to recognize that meaning, and continue using the word to mean, "Someone who loves to program and enjoys being clever about it."

Stallmanin määritelmä koskee lähinnä tietokonehakkereita, mutta hakkeri voi tarkoittaa myös muun alan innokasta harrastajaa. Williams [72] selostaa käsitettä tarkemmin B-liitteessään (Hack, Hackers, and Hacking), mutta Stallmanin antama määritelmä on riittävä tässä vaiheessa.

Hakkerikulttuurin alku voidaan sijoittaa vuoteen 1961, jolloin *MIT (Massachusetts Institute of Technology)* hankki ensimmäisen PDP-1-tietokoneensa [55][2]. MITissä toi-

[1] Lainaus Emacs-editorin mukana tulevasta tiedostosta THE-GNU-PROJECT.

[2] Eräs Digital Electronic Corporationin valmistama tietokonemalli. PDP tuli sanoista Programmed Data Processor. Huhun mukaan DEC ei halunnut nimetä laitettaan tietokoneeksi, koska tuohon

mineen pienoisrautateihin keskittyneen harrastuskerhon, *TMRC:n (Tech Model Railroad Club)*, alajaosto Signals and Power committee otti sen innokkaasti käyttöönsä ja kehitti siihen erilaisia ohjelmia. Tästä syntyi vähitellen oma kulttuuri, jota mm. Steven Levy on kuvannut kirjassaan [39]. Tämän uuden kulttuurin edustajista, innokkaista tietokoneharrastajista, alettiin käyttää nimitystä hakkeri.

Hakkerikulttuurin keskuksena toimi MITin tekoälylaboratorio. [72] Laboratoriossa käytetty *ITS-käyttöjärjestelmä (Incompatible Timesharing System)* oli henkilökunnan alusta alkaen itse suunnittelema ja toteuttama, kuten myös moni muukin ohjelma. [62, 72] Tietokoneohjelmia jaeltiin täysin vapaasti. Tämä ei ollut mitenkään erikoista, sillä tätä tapaa oli noudatettu jo aivan tietokoneajan alusta lähtien. [68] Hakkerikulttuurissa jakamista harrastettiin vain enemmän kuin muualla, sillä siellä uusia ohjelmia syntyi nopeaan tahtiin. Näitä ohjelmia ei kuitenkaan nimitetty "vapaiksi ohjelmiksi", koska kyseistä termiä ei vielä ollut olemassa. Käytännössä kyse oli kuitenkin juuri niistä. [17, 62]

3.2 Vapaat ohjelmat

Hakkerikulttuuri kukoisti. Kuka tahansa tietokoneista kiinnostunut sai tulla käyttämään arvokkaita tietokoneita. Hakkerit eivät kysyneet tulijoilta, oliko heillä lupa käyttää tietokoneita, vaan osasivatko he käyttää niitä[3]. Tämän vapaan, ehkä hieman anarkistisen, ilmapiirin päälle alkoi kuitenkin kasaantua tummia pilviä[4].

MITissä oli kehitetty uudenlainen tietokonejärjestelmä, joka käytti kaikkialla Lisp-ohjelmointikieltä aina laiteajureista varusohjelmistoon asti. [72] Tämän *Lisp Machine* -järjestelmän pääarkkitehti Richard Greenblatt halusi perustaa ns. hakkeriystävällisen yrityksen, joka valmistaisi ja myisi tietokonejärjestelmiä. Koska Greenblattin kaupallisia kykyjä epäiltiin, hän palkkasi Russell Noftskerin hoitamaan perusta-

aikaan tietokoneiden "tiedettiin" olevan isoja ja kalliita.
http://www.oldcomputers.arcula.co.uk/pdp81.htm (viitattu 26.9.2007)

[3] "HACKERS SHOULD BE JUDGED BY THEIR HACKING, NOT BOGUS CRITERIA SUCH AS DEGREES, AGE, RACE, OR POSITION." [39]

[4] Seuraavassa esitetyt historialliset tiedot ovat peräisin lähdekirjallisuudesta, eikä niiden todenperäisyyttä ollut kirjoitushetkellä kyseenalaistettu missään. 11.11.2007 Dan Weinreb, yksi Symbolicsin perustajista, kuitenkin julkaisi verkkopäiväkirjassaan kirjoitelman otsikolla "Rebuttal to Stallman's Story About The Formation of Symbolics and LMI", jonka mukaan Stallman on vääristellyt historiaa omaksi edukseen. Tilanne vaatisi lisäselvityksiä, joita ei kuitenkaan ole mahdollista tehdä tämän tutkielman puitteissa. http://dlweinreb.wordpress.com/2007/11/11/rebuttal-to-stallmans-story-about-the-formation-of-symbolics-and-lmi/ (viitattu 15.11.2007).

mansa yrityksen *LMI:n (Lisp Machines, Incorporated)* talousasioita.

Noftsker ei pitänyt Greenblattin liikeideaa toimivana, joten hän perusti kilpailevan yrityksen, *Symbolics*in. Symbolics erosi LMI:stä siinä, että LMI:n ideana oli toimia ilman ulkopuolisia investointeja, kun taas Symbolics alkoi heti alusta alkaen hankkia sijoituksia ulkopuolisilta rahoittajilta.

Symbolics pyrki alusta asti saavuttamaan etumatkan kilpailijaansa nähden. Alun perin kaikki kolme Lisp Machine -järjestelmää kehittävää tahoa, MIT, LMI ja Symbolics, jakoivat tekemänsä muutokset toisilleen, mutta tilanne muuttui, kun Symbolics kielsi omien muutostensa käytön toisissa järjestelmissä. [72]

Tämän lisäksi Symbolics alkoi palkata hakkereita oman järjestelmänsä kehitykseen. Moni siirtyikin yrityksen palvelukseen, mutta joutui samalla luopumaan ohjelmien vapaan levityksen filosofiastaan. [17]

Tämä oli suuri muutos kaikille, jotka olivat pitäneet lähdekoodien vapautta itsestäänselvyytenä. Heidän mukaansa oli käsittämätöntä, että heiltä otettiin pois mahdollisuus muokata ohjelmia haluamikseen ja, mikä vielä pahempaa, mahdollisuus antaa omat muutokset toisten käyttöön.

Yksi näistä ohjelmien mukauttamiseen ja levittämiseen tottuneista henkilöistä oli Richard Stallman. Stallman oli tullut MITin tekoälylaboratorioon vuonna 1971. [62] Hänen tehtävänään oli mm. ITS:n kehittäminen, mutta hän oli saavuttanut mainetta myös *EMACS*-tekstieditorillaan. Nyt hän tunsi, että koko se hakkeriyhteisö, johon hän kuului, oli luhistumassa. Hän mietti vaihtoehtojaan: loistavana ohjelmoijana[5] hän voisi siirtyä jonkin kaupallisen yrityksen palvelukseen jatkaakseen ohjelmointiuraansa. Tämä kuitenkin tarkoittaisi toisten hakkerien pettämistä. Stallman harkitsi jopa siirtymistä pois ohjelmoinnin parista.

Aluksi Stallman pysyi puolueettomana kahden yrityksen kilpailun keskellä. Kun lopulta tuli mahdottomaksi pysyä ulkopuolisena, hän päätyi auttamaan LMI:tä pitämään pintansa Symbolicsia vastaan. [72] Olihan Symbolics toiminnallaan edistänyt hakkerikulttuurin luhistumista.

Näkemyksen Stallmanin mietteistä antaa seuraava lainaus ZMail-ohjelman ohjekirjasta[6]:

> The author [...] believes that the commercialization of the computer industry hinders the further development of systems such as described herein. He considers proprietary software morally objectionable and plans

[5] [68] käyttää termiä "brilliant programmer".

[6] ZMail Manual, First Edition, ZMail Version 50 (System 94), April 1983.

> to dedicate his career to promoting the sharing and free exchange of software.

Jonkin aikaa mietittyään Stallman päätyi merkittävään lopputulokseen. Hän päätti herättää hakkeriyhteisön uudelleen henkiin. Hän asetti itselleen myös uuden päämäärän. Koska ohjelmistojen kaupallistuminen oli aiheuttanut hakkeriyhteisön hajoamisen, hän päätti luoda kokonaisen vapaan käyttöjärjestelmän, aina ytimestä sovellusohjelmiin asti. Tämän järjestelmän kaikki osat olisivat täysin vapaat, eli kuka tahansa voisi muokata niitä ja jakaa eteenpäin. Tälle vapaan käyttöjärjestelmän projektilleen Stallman antoi nimeksi hauskan rekursiivisen akronyymin, *GNU:n (GNU's Not Unix)*. [17, 62]

Vuonna 1984 Stallman kertoi tavoitteestaan Lisp Machinen ohjekirjassa seuraavasti[7]:

> I believe that the commercialization of computer software has harmed the spirit which enabled such systems to be developed. Now I am attempting to build a software-sharing movement to revive that spirit from near oblivion.
>
> Since January 1984 I have been working primarily on the development of GNU, a complete Unix-compatible software system for standard hardware architectures, to be shared freely with everyone just like EMACS. This will enable people to use computers and be good neighbors legally (a good neighbor allows his neighbors to copy any generally useful software he has a copy of).

Stallman lopetti työskentelynsä MITissä ennen projektinsa aloittamista, koska se oli ainoa tapa, jolla hän pystyi päättämään itse tekemiensä ohjelmien jakeluehdoista. [72] Mikäli hän olisi jäänyt MITiin, se olisi voinut vaatia omaa osaansa työntekijänsä työn tuloksista. Lopettamisensa jälkeen Stallman sai kuitenkin jatkaa MITin tilojen ja laitteiden käyttämistä tekoälylaboratorion johtajan professori Winstonin suostumuksella. [62]

Stallmanin mukaan ohjelma on vapaa, jos se täyttää seuraavat ehdot [62]:

1. Kuka tahansa saa käyttää ohjelmaa mihin tarkoitukseen tahansa.
2. Kuka tahansa saa tutkia ohjelman toimintaa ja muuttaa sitä millä tahansa tavalla.

[7] Lisp Machine Manual, Sixth Edition, System Version 99, June 1984.

3. Kuka tahansa saa levittää ohjelmaa eteenpäin. Levittämisen voi tehdä ilmaiseksi tai siitä saa ottaa maksun.

4. Kuka tahansa saa parannella ohjelman toimintaa ja antaa tekemänsä muutokset eteenpäin.

Stallmanin tavoitteena oli, että GNU olisi mahdollisimman vapaa. Hän huomasi, että pelkkä ohjelman vapaa jakelu ei ollut riittävää. Jos hän jakaisi ohjelmaa esimerkiksi *PD*-tyyppisesti *(Public Domain)*, kuka tahansa voisi ottaa ohjelman ja muuttaa sen epävapaaksi eli *omisteiseksi* (engl. proprietary). Estääkseen tämäntyyppisen tilanteen synnyn Stallman kirjoitti uudenlaisen ohjelmistolisenssin, *GNU GPL:n (General Public License),* joka takasi sen, että sen ehtojen mukaan levitetyt ohjelmat olivat vapaita ja myös pysyisivät vapaina [62, 72][8]. Lisenssi on "tarttuva", eli sen ehdot laajenevat koskemaan jokaista sen alaista koodia hyödyntävää sovellusta. Taktisista syistä myöhemmin julkaistiin myös hieman lievennetty *LGPL (Lesser General Public License),* joka on muuten samanlainen kuin GPL, mutta ei tartu.

Hankkiakseen varoja GNU-projektille Stallman ja muut projektiin liittyneet kehittäjät perustivat *Free Software Foundationin (FSF)* vuonna 1985. [17, 62, 72] Nykyään FSF myy GNU-projektin ohjelmia ja painettuja ohjekirjoja. FSF myös omistaa tekijänoikeudet suureen osaan GNU-projetin ohjelmista[9].

3.3 Linux

Vuoteen 1991 mennessä GNU-projekti oli edennyt jo siihen pisteeseen, että suurin osa käyttäjätilan ohjelmista oli valmiina. Suurin merkittävä puute oli käyttöjärjestelmän ydin. Se oli myös tärkein osa, sillä ilman ydintä käyttöjärjestelmä ei toimi. Käyttäjätilan ohjelmia tosin pystyi käyttämään jo olemassa olevissa järjestelmissä, mutta se ei riittänyt. Koko järjestelmä ei ollut vielä vapaa.

Samana vuonna Linus Torvalds, parikymppinen tietojenkäsittelytieteen opiskelija Helsingin yliopistossa, aloitti oman käyttöjärjestelmän teon. Alun perin hän aikoi tehdä yksinkertaisen pääteohjelman, jolla voisi olla yhteydessä yliopiston tieto-

[8] Ensimmäinen GPL:n versio julkaistiin helmikuussa 1989, toinen kesäkuussa 1991 ja kolmas kesäkuussa 2007. Nykyään suurin osa kaikista vapaista ohjelmista käyttää GPL:ää.

[9] "Some GNU software is written by staff of the Free Software Foundation, but most GNU software is contributed by volunteers. Some contributed software is copyrighted by the Free Software Foundation; some is copyrighted by the contributors who wrote it."
http://www.gnu.org/philosophy/categories.html (viitattu 27.7.2007).

koneisiin. Vähitellen hän lisäsi ohjelmaansa uusia toimintoja, kunnes se sisälsi suurimman osan käyttöjärjestelmäytimen toiminnoista.

Aluksi Linus jakoi tätä Linuxiksi nimettyä ydintä[10] omatekoisen lisenssin alaisuudessa, mutta vaihtoi tämän pian GPL:ksi. Kun GNU-projektin toteuttamat ohjelmat ja Linusin Linux-ydin yhdistettiin, muodostui toimiva käyttöjärjestelmä, joka oli täysin vapaa.

Tämä Linuxina tunnettu käyttöjärjestelmä on nykyään erittäin suosittu, mutta suosion varjopuolena on se, että monikaan ei muista sen historiaa. Erityisesti GNU-projektin merkitys jää usein mainitsematta. Tilannetta on pyritty parantamaan kutsumalla järjestelmää nimellä GNU/Linux ja selostamalla, että Linux on vain käyttöjärjestelmän ydin, sitä kuitenkaan aliarvioimatta. Valitettavasti tämä toiminta on usein ymmärretty väärin GNU-projektin pyrkimykseksi saada itselleen kunniaa jostain sellaisesta, joka ei sille kuulu.

Voisi sanoa, että Linux-ytimen myötä vapaat ohjelmistot pääsivät niille kuuluvaan asemaan. Syynä oli se, että vapaat ohjelmat olivat ennen Linuxin tuloa olleet suuren yleisön tietämättömissä. Linuxin tultua kuka tahansa pystyi asentamaan valtavan määrän vapaita ohjelmia omalle tietokonelleen. Aika oli muutenkin sopiva Linux-pohjaisille järjestelmille. Monen mielestä Microsoftin Windows-käyttöjärjestelmälle oli viimeinkin löytynyt riittävän tasokas haastaja.

3.4 Avoin lähdekoodi

Avoin lähdekoodi on ilmiö, jonka todellisesta luonteesta ei usean vuoden jälkeenkään ole päästy täyteen selvyyteen. Joidenkin mielestä kyseessä on joukko uudenlaisia ohjelmistojen kehitysmenetelmiä [59], mutta toiset pitävät sitä vain lisenssikysymyksenä [68]. Yhden näkemyksen mukaan avoin lähdekoodi rikkoo tunnettuja ohjelmistokehityksen lakeja [54], mutta toisen mukaan se ei tarjoa mitään uutta[11]. Erilaisia näkemyksiä riittää siis joka lähtöön, joten mitä tahansa käsitystä voi uskottavasti perustella viittaamalla johonkin sopivaan lähteeseen.

[10] Alun perin Linus kutsui tuotostaan nimellä Freax, mutta Funetin FTP-palvelimen ylläpitäjä Ari Lemmke ei pitänyt siitä, vaan antoi sille nimen Linux.

[11] "Open source is nothing new. I clearly remember when in 1976 I started working with Unix [...] It felt like absolute freedom. I was able to [...] do whatever I wanted with Unix [...] All this because I had the sources of Unix, written in this strange language called C [...] I don't think that I exaggerate when I say that Open Source has been an integral part of the technology revolution of the last 30+ years." http://www.jroller.com/peterz/entry/b_the_open_source_poison (viitattu 5.11.2007).

Tässä tutkielmassa ei oteta kantaa siihen, mikä näkemys on mahdollisesti oikea. Asiaa esitellään sekä avoimen lähdekoodin merkkihenkilöiden näkemysten että erilaisten tutkimusten valossa, jolloin lukija voi tehdä asiasta omat johtopäätöksensä. Mutta kuten Fogel ja Bar toteavat, aihe on jo niin laaja, ettei yksikään ihminen voi käsittää sitä kokonaisuudessaan. [17]

3.4.1 Eric S. Raymondin näkemys

Avoimen lähdekoodin käsitteen luojana voidaan pitää Eric S. Raymondia, joka edellä mainitusta Linux-ilmiöstä kiinnostuneena ja sitä tutkittuaan laati havainnoistaan tunnetun kirjoitelmansa *The Cathedral and the Bazaar*[12]. Tässä Linux-ilmiön analyysissään Raymond käsittelee kahta vapaiden ohjelmien kehitysmenetelmää, katedraalia ja basaaria. [54] Alun perin hän tarkoitti niillä lähinnä GNU-projektia ja Linux-kehitystä, mutta myöhemmin ne muuntuivat kuvaamaan suljettua ja avointa kehitystä. Raymondin mukaan Linuxissa käytetty kehitysmenetelmä vastaa vilkasta basaaria, jossa jokainen voi antaa oman panoksensa yhteisön hyväksi. Tätä vastoin esimerkiksi Emacs-editorin kehitys muistuttaa katedraalin rakentamista siinä, että siihen ottavat osaa vain todelliset ammattilaiset, eikä siitä julkaista keskeneräisiä versioita ulkopuolisten käyttöön.

Raymondin kirjoitusten innostamana Netscape julkaisi oman www-selaimensa lähdekoodin vuonna 1999. [17] Tämän tapahtuman ja Linuxin saavuttaman suuren suosion vuoksi myös muut kaupalliset yritykset kiinnostuivat Raymondin esittämistä ideoista, etenkin Linuxin yhteydessä toimivaksi osoittaneesta kehitystavasta. Jonkin verran epäröintiä aiheutti kuitenkin englannin kielen sana “free”, mikä tarkoittaa sekä vapaata että ilmaista.

3.4.2 Bruce Perensin näkemys

Koska Raymondin Linux-yhteisön piiristä “löytämä” kehitysmenetelmä ei olisi toiminut suljetussa ympäristössä, pidettiin selvänä, että siitä kiinnostuneet olisivat valmiita vapauttamaan ohjelmansa yhteisön käyttöön. Ainoa mitä tarvitsi tehdä, oli saattaa yritykset tietoisiksi niistä eduista, joita lähdekoodin vapauttaminen toisi tullessaan. Tämä merkitsi vapaiden ohjelmien kannattajille mahdollisuutta saada äänensä kuuluviin.

[12] http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/ (viitattu 24.9.2007).

Yritysten mielenkiinnon herätessä selvisi pian, ettei käsite "free software" ollut riittävän tunnettu, joten sen uskottiin tarkoittavan ilmaisia ohjelmia. Tämän virhekäsityksen ehkäisemiseksi vapaiden ohjelmien markkinoinnissa otettiin käyttöön uusi nimitys *Open Source*, eli "avoimen lähdekoodin ohjelmat", jolla korostettiin sekä avoimuutta (joka nähtiin korvikkeena vapaudelle) että sitä, että varsinainen tuote on sovelluksen lähdekoodi, eikä perinteisesti ajateltu binääriversio.

Raymondin ehdotuksesta mm. omaa Linux-jakelua kehittävästä Debian-projektista[13] tunnetuksi tullut vapaiden ohjelmien puolestapuhuja Bruce Perens laati avoimen lähdekoodin ohjelmille seuraavat periaatteet[14]:

1. Avoimen lähdekoodin alaista ohjelmaa voidaan levittää maksua vastaan tai ilmaiseksi.

2. Ohjelman lähdekoodin pitää olla saatavilla. Parhaassa tapauksessa sen saa maksutta, mutta pieni maksukin on sallittu esim. tallennusmedian hinnan ja postikulujen kattamiseksi.

3. Lähdekoodin tutkimiselle ja muuttamiselle ei saa asettaa rajoituksia.

4. Mikäli ohjelmaa muutetaan, muutetulle ohjelmaversiolle voidaan asettaa joitakin rajoituksia. Esimerkiksi voidaan vaatia, että muutetun version nimi muutetaan joksikin toiseksi (vrt. StarOffice vs. OpenOffice), tai että ohjelma pitää jakaa siten, että alkuperäinen lähdekoodi pidetään muuttamattomana ja sen mukana annetaan erillinen muutostiedosto (ns. patch file).

5. Avoimen lähdekoodin lisenssi ei saa olla ketään syrjivä.

6. Kenellä tahansa on oikeus käyttää ohjelmaa.

7. Ohjelman käyttöoikeuksien on oltava kaikille samat. Kenellekään ei voi antaa erityisoikeuksia, eikä kenellekään voi asettaa erityisrajoituksia.

8. Lisenssi ei voi rajoittua vain tiettyyn tuotteeseen. Jos ohjelma on esim. mukana jollain kokoelma-CD:llä, se pitää voida ottaa siitä erilleen levitykseen saman lisenssin alaisuudessa.

[13] http://www.debian.org/ (viitattu 29.10.2007).
[14] http://www.opensource.org/docs/osd/ (viitattu 29.10.2007).

9. Lisenssi ei vaikuta ulkopuolisiin ohjelmiin. Mikäli avoimen lähdekoodin ohjelma on mukana esim. kokoelma-CD:llä, lisenssi ei koske muihin saman CD:n ohjelmia.

10. Lisenssi ei riipu käytettävästä teknologiasta. Jos esimerkiksi ohjelman voi ladata Internetistä suoraan tietokoneelle, mutta sen voi myös tilata erillisenä tuotteena postin välityksellä, saapuneen paketin avaaminen ei merkitse lisenssin hyväksymistä.

Vuonna 1998 avoimen lähdekoodin asiaa ajamaan ja näiden periaatteiden noudattamista valvomaan perustettiin *Open Source Initiative* -niminen organisaatio[15], jonka edustama asia kiteytetään sen www-sivustolla seuraavasti: "The basic idea behind open source is very simple: When programmers can read, redistribute, and modify the source code for a piece of software, the software evolves. People improve it, people adapt it, people fix bugs. And this can happen at a speed that, if one is used to the slow pace of conventional software development, seems astonishing."

Raymondin ja Perensin yhteistyöstä huolimatta viimeistään vuoden 1999 helmikuussa tuli selväksi, että heidän näkemyksensä avoimesta lähdekoodista poikkesivat toisistaan. Perensin Debian-kehityslistalle lähettämästä viestistä voidaan havaita, miten hän oli pitänyt avointa lähdekoodina tapana esitellä vapaat ohjelmat suurelle yleisölle[16]:

> Eric Raymond and I founded the Open Source Initiative as a way of introducing the non-hacker world to Free Software. [...] Now that the world is watching, it's time for us to start teaching them about Free Software.
>
> Most hackers know that Free Software and Open Source are just two words for the same thing. Unfortunately, though, Open Source has deemphasized the importance of the freedoms involved in Free Software. It's time for us to fix that.

Perens jatkoi selostamalla, miten avoimen lähdekoodin liikkeen oli tarkoitus olla yhteensopiva vapaiden ohjelmien ideologian kanssa. Hänen mukaansa Eric S. Raymond oli kuitenkin unohtanut alkuperäisen tavoitteen:

> Sadly, as I've tended toward promotion of Free Software rather than Open Source, Eric Raymond seems to be losing his free software focus.

[15] http://www.opensource.org/ (viitattu 16.1.2007).

[16] http://lists.debian.org/debian-devel/1999/02/msg01641.html (viitattu 23.8.2007).

> [...] I fear that that the Open Source Initiative is drifting away from the Free Software values with which we originally created it. [...]

Yhtenä ristiriitojen aiheuttajana voidaan pitää sitä, että Perensin näkemys avoimesta lähdekoodista vastasi niin paljon Stallmanin kantaa, ettei se kelvannut Raymondille, ja niin paljon Raymondin kantaa, ettei se kelvannut Stallmanille.

3.4.3 Richard Stallmanin käsitys

Raymondin ja Perensin väliset näkemyserot eivät olleet mitään uutta, sillä ristiriitoja oli esiintynyt heti avoimen lähdekoodin liikkeen alkuaikoina, kun sen idea oli ymmärretty väärin ja sitä oli alettu pitää vapaiden ohjelmien liikkeen kilpailijana. Bruce Perens kertoo asiasta näin [51]:

> Richard Stallman later took exception to the campaign's lack of emphasis of freedom, and the fact that as Open Source became more popular, his role in the genesis of free software, and that of his Free Software Foundation, were being ignored–he complained of being "written out of history."

Vapaiden ohjelmien ja avoimen lähdekoodin liikkeiden kannattajien väliset erimielisyydet ovat yleisiä vielä nykyäänkin. Näitä kiistoja seuratessa käy kuitenkin nopeasti selväksi, ettei juuri kukaan ole oikeasti viitsinyt perehtyä asioihin. Erikoista kuitenkin on, etteivät edes liikkeiden kannattajat tunnu huomaavan sitä, että avoimella lähdekoodilla tarkoitetaan nykyään itse asiassa kahta eri asiaa:

1. Avoimen lähdekoodin ohjelmilla (Open Source Software) tarkoitetaan vapaita ohjelmia (Free Software), mutta sekaannusten välttämiseksi (vapaa vai ilmainen?) käytetään eri termiä.

2. Avoimen lähdekoodin ohjelmilla (Open Source Software) tarkoitetaan kaikkia niitä ohjelmia, jotka on tehty basaari-tyyppisen kehitysmenetelmän mukaisesti. Tämän menetelmän toimivuus kuitenkin vaatii sen, että ohjelman lisenssi täyttää annetut ehdot.

Vaikka nämäkään määritelmät eivät ole yleisesti hyväksyttyjä, ne voivat osaltaan selittää sitä, miksi joidenkin mielestä liikkeillä ei ole mitään eroja (näkemys 1) ja miksi toiset näkevät avoimen lähdekoodin liikkeen joko lisenssi- tai kehitysmenetelmäkysymyksenä (näkemys 2).

3.4.4 Tieteellisiä näkemyksiä

Käytännössä avoimen lähdekoodin eri vivahteiden ja vapaiden ohjelmien välisten erojen ymmärtäminen on osoittautunut niin vaikeaksi, etteivät edes kaikki tutkijat osaa erotella niitä toisistaan. Thomas Østerlie ja Letizia Jaccheri kritisoivatkin aihetta käsittelevää kirjallisuutta siitä, että siinä yleensä niputetaan kaikki aihepiiriin liittyvät ilmiöt yhteen, eikä niiden välisiä eroja juurikaan huomioida. [74].

Lin [40] toteaa, että käsite "avoin lähdekoodi" on yleistyessään menettänyt jonkin verran tarkkuuttaan. Hän muistuttaa myös, että eri henkilöt voivat antaa samoille käsitteille eri tulkintoja.

Elliott [14] toteaa aiheeseen liittyen, että tiedeyhteisö on huomannut käsitteiden erot:

> However, the research community recognizes differences between the meanings of these terms. The research literature often refers to "free/open source software" (F/OSS) to refer to both types of software together.

Tämä käsitteiden niputtaminen onkin ymmärrettävää silloin, kun erolla ei ole mitään vaikutusta tutkittavaan aiheeseen. Haittapuolena voi kuitenkin olla, että tutkimuksen tuloksista tehdään niin pitkälle meneviä päätelmiä, että käsitteiden samaistaminen johtaa virheellisiin tulkintoihin.

Yhtenä esimerkkinä käsitteiden yhdistämisestä voidaan pitää Scacchin käsitystä [59]:

> [Free/Open Source Software Development] is a different, somewhat orthogonal approach to the development of software systems where much of the development activity is openly visible, development artifacts are publicly available over the Web, and generally there is no formal project management regime, budget or schedule.

Lainaus keskittyy kuvaamaan ohjelmistokehityksen lähestymistapaa, joten vapaiden ohjelmien merkitys jää hieman epäselväksi.

Sama huomio toistuu myös Gacekin ja Ariefin [20] kohdalla. Vaikka he onnistuvatkin ansiokkaasti tuomaan esille avointen kehitysprojektien toimintakulttuuria, he jättävät Stallmanin näkemyksen vapaiden ohjelmien sosiaalisesta merkityksestä kokonaan huomiotta.

Niissä tutkimuksissa, joissa käytetään pelkästään termiä "avoin lähdekoodi", käsitteen katsotaan tarkoittavan ainoastaan lisensointitapaa. Esimerkiksi Warsta toteaa näin [70]:

> OSS is not a compilation of well defined and published software development practices constituting an eloquent, written method. Instead, it is better described in terms of different licenses for software distribution and as a collaborative way of widely dispersed individuals to produce software with small and frequent increments.

Myös von Hippel on samoilla linjoilla. Hän tosin myöntää, että mukana on myös erilainen ohjelmistonkehitysmenetelmä [68]:

> [Open source] denotes only the type of license under which it is made available. However, the fact that open source software is freely accessible to all has created some typical open source development practices that differ greatly from commercial software development models—and that look very much like the "hacker culture" behaviors[...]

Oezbekin ja Precheltin mielestä projektia ei voida pitää tyypillisenä avoimen lähdekoodin projektina ainoastaan lisenssin perusteella, vaan he vaativat siltä seuraavaa [48]:

> [P]roject should be Open Source not only by license but also by development style: The project members need to be distributed rather than co-located at a single company site, communication must be public and preferably archived, it must be possible for external newcomers to join the project, and basic processes and tools (such as release process, issue tracker and version repository) should be established.

Kaikesta tästä voi päätellä, että myöskään tieteellisessä tutkimuksessa ei ole yhtä oikeaa määritelmää avoimelle lähdekoodille. Tämä ei kuitenkaan ole estänyt tutkimusten tekoa, kunhan käsiteltävän aiheen ei ole tarvinnut ottaa kantaa näkemyseroihin.

3.5 Avoimen lähdekoodin prosessimalli

Tietotekniikan alkuaikoina ohjelmistokehitys tarkoitti sitä, että ohjelmoija kirjoitti koodia, joka ratkaisi jonkin ongelman. Aikojen saatossa ohjelmistojen koko on kuitenkin kasvanut ja ne ovat muuttuneet niin monimutkaisiksi, ettei yhdenkään ihmisen uskota enää yksinään voivan hallita kokonaisuutta. Tämän vuoksi nykyään

kehityksestä vastaavatkin kehitystiimit, joissa voi olla analyytikoita, ohjelmistoarkkitehtejä, ohjelmoijia, testaajia jne., joista jokainen on erikoistunut johonkin tiettyyn osa-alueeseen.

Nykyään ohjelmistot voivat koostua miljoonista koodiriveistä ja kehitystiimit kymmenistä asiantuntijoista. Tilanne on sama sekä teollisuudessa että avoimen lähdekoodin puolella, joten molemmilla puolilla on periaatteessa samat ongelmat ratkaistavana. Ratkaisut ovat kuitenkin hieman erilaisia.

Ohjelmistoteollisuudessa kokonaisuuden hallintaan on käytetty erilaisia kehitysmalleja, joiden mukaan toimimalla asiat ovat edistyneet hallitusti. Avoimen lähdekoodin projekteissa ei taas ole käytetty mitään perinteisiä malleja, vaan projekti on edennyt omalla painollaan.

Pressmanin [53] mukaan ohjelmistokehitysprojektissa käytettävän prosessimallin valintaan vaikuttavia seikkoja ovat

- projektin ja kehitettävän sovelluksen luonne,
- käytettävät työkalut ja -menetelmät,
- vaadittava kontrollin määrä.

Avoimen lähdekoodin projektien ja perinteisten ohjelmistoprojektien erojen sanotaan johtuvan niiden käyttämistä prosessimalleista. Tämä pitää jossain määrin paikkansa, mutta sen ei voi katsoa olevan perimmäinen syy. Todennäköisempi syy eroavuuksille löytyy ylläesitetyistä prosessimallin valintaan vaikuttavista seikoista.

Avoimen lähdekoodin projektien luonne eroaa melkoisesti perinteisistä ohjelmistoprojekteista. Eräänä syynä on luonnollisesti se, että avoimen lähdekoodin projekteista suuri osa on vapaaehtoispohjalla toimivia, kun taas perinteiset projektit liittyvät tavallisesti kaupalliseen toimintaan.

Projektien koko voi vaihdella huomattavasti yksittäisistä kehittäjistä tuhansiin. Monen kehittäjän projekteissa kommunikaation merkitys korostuu, ja projekti organisoidaan eri tavalla, kuin pienemmän mittakaavan projektit. Kehittäjien sijoittuminen ympäri maapalloa tuo omat haasteensa.

Projektien tavoitteet ovat myös toisistaan poikkeavia. Avoimen lähdekoodin projekteilla on yleensä lyhyen aikavälin tavoitteita, eikä pidemmän ajan suunnitelmia juuri ole. Kaupallisissa projekteissa tavoitteena on täyttää vaatimusmäärittelyyn kirjatut tavoitteet.

Perinteisissä ohjelmistoprojekteissa käytetyt työkalut on yleensä määritelty etukäteen. Avoimen lähdekoodin projekteissa näin ei ole, vaan kuka tahansa voi käyttää mitä tahansa kehitystyökaluja, kunhan lopputulos on sopiva.

Ehkä suurin ero avoimen lähdekoodin projektien ja perinteisten ohjelmistoprojektien välillä on se, kuinka paljon prosessia pitää hallita. Perinteisissä ohjelmistoprojekteissa projektinhallinta on tärkeä osa-alue, koska sen avulla onnistuneiden projektien oppeja voidaan käyttää myös tulevaisuudessa. Olemassa olevien resurssien asettamien raamien sisällä pysyminen on helpompaa, kun tiedetään miten projekti etenee. Avoimen lähdekoodin projekteissa kontrolli on vähäisempää, koska taloudellisia riskejä ei yleensä ole. Myöskään aikarajat eivät ole niin joustamattomia kuin perinteisesti.

Yllä olevien erojen pohjalta on mahdollista sanoa, että avoimen lähdekoodin ja perinteisten ohjelmistoprojektien väliset erot johtuvat pikemminkin projektien luonteista kuin käytetyistä kehitysmenetelmistä. Silmiinpistävimpänä erona näkyy käytössä olevien resurssien aiheuttamat vaatimukset prosessin kontrollointiin. Voisi olettaa, että eri projekteja ei voisi erottaa toisistaan, mikäli kummallakin olisi samat rajoitukset tai vapaudet resurssien suhteen.

Siihen, kumpi tapa on parempi, ei voi vastata lopullisesti. Molemmissa on hyvät ja huonot puolensa. Tulosten laadukkuuden puolesta voisi sanoa, että avoimen lähdekoodin projektit voivat olla huomattavasti laadukkaampia, mutta myös huonompia. Perinteisten ohjelmistoprojektien tulokset, mikäli niitä saadaan, ovat yleensä tasalaatuisia.

Avoimen lähdekoodin projektit ovat hyvin monimuotoisia, vaihdellen yksittäisten ohjelmoijien viikonloppuharrastuksista tuhansien kehittäjien projekteihin. Suurimmalla osalla projekteista on kuitenkin joitain yhteisiä piirteitä. Seuraavassa listassa esitellään Johnsonin [35] löytämiä piirteitä. Suomennokset ovat melko vapaita ja niitä on täydennetty asian valaisemiseksi.

1. Projekti alkaa yleensä vain yhden kehittäjän tai pienen kehitysryhmän voimin. Alussa kehitettävästä sovelluksesta toteutetaan vain yksinkertainen versio tai prototyyppi, josta voidaan päätellä, onko tavoite mahdollista toteuttaa. Mikäli projekti nähdään kehityskelpoiseksi, se laitetaan julkisesti nähtäville ja kehitystä jatketaan. Mikäli aihe on kiinnostava, projektin ympärille muodostuu kiinnostuneiden kehittäjien muodostama virtuaalinen yhteisö.

2. Projektissa käytetään iteratiivista ja inkrementaalista prosessimallia. Kehityk-

sen kohteena olevaa koodia muutetaan vähän kerrallaan. Jokainen muutos pyritään testaamaan aikaisessa vaiheessa, minkä jälkeen palataan uusien muutosten tekoon. Muutokset ovat yleensä pieniä, mutta niitä tehdään nopeassa tahdissa.

3. Kehitysprosessi on hajautettua. Koska kehittäjät eivät ole fyysisesti sidottuja mihinkään aikaan ja paikkaan, koodia voidaan muuttaa milloin tahansa. Tämä saattaa johtaa siihen, että koodi on epävakaata jonkin ajan. Koska myös testaus on hajautettua, epävakaus ei yleensä kestä pitkää aikaa, ennen kuin joku löytää epävakauden aiheuttajan ja korjaa sen.

4. Koodin katselmointi on erittäin laaja-alaista. Lähes kaikki muutokset tarkastetaan usean kehittäjän voimin ennen niiden käyttöönottoa. Tämän vuoksi suurin osa ongelmista havaitaan jo ennen kuin ne ehtivät aiheuttaa ongelmia. Koska koodin katselmointiin osallistuu useita henkilöitä, tarkastelu tapahtuu usean eri näkökulman, kuten esim. koodin siisteyden, tietoturvan, dokumentoinnin ja tehokkuuden kannalta.

5. Koodiin tehdään muutoksia tarpeen mukaan. Sellaisia ominaisuuksia, joille ei ole selvää tarvetta, ei yleensä tehdä. Koska kehittäjät ovat yleensä itse myös käyttäjiä, tehdyt muutokset ovat tavallisesti oikeansuuntaisia ja täyttävät niille asetetut tavoitteet. Laajoja tarvekartoituksia ei myöskään tarvitse tehdä.

6. Toiminta on hajautettua. Kun projektin koko kasvaa, eri osa-alueet voivat muodostua omiksi aliprojekteiksi. Aliprojekteilla voi olla erilliset vastuuhenkilöt, jotka huolehtivat omasta kokonaisuudestaan. Aliprojektit toimivat kuitenkin yhdessä kokonaisprojektin alaisuudessa ja niiden toimintaa koordinoidaan koko projektin tasolla.

7. Projektin johtajuus perustuu luottamukselle. Kehittäjät, joihin luotetaan, saavat suurimman osan päätösvallasta ja samalla vastuusta. Heidän näkemyksensä on merkittävämpi kuin muiden projektiin osallistuvien. Luottamus on kuitenkin ansaittava osoittamalla taitonsa omalla toimialallaan. Kun luottamus on ansaittu, sitä on pidettävä yllä jatkuvalla toiminnalla, sillä pelkkä aikaisemmin hankittu maine ei riitä, kun mukaan tulee uusia kehittäjiä, joihin vanhat uroteot eivät tehoa.

8. Projektiin osallistumisen motiivit ovat yleensä henkilökohtaisia. Tavallisesti osallistumisen syynä ei ole raha tai mikään ulkonainen syy. Joillekin syynä on

halu tehdä jotain oman yhteisön eteen, toisia kiinnostaa mahdollisuus saada mainetta ja kunniaa, ja tuoda esille omaa osaamistaan. Toisille koko projekti voi olla jonkinlaista roolipeliä, jossa tavoitteena on nousta arvohierarkiassa mahdollisimman korkealle. Joissain tapauksissa kehittäjän työnantajalla voi olla intressejä projektiin, jolloin työntekijä voi saada palkkaa kyseiseen projektiin osallistumisesta.

9. Koska projektiin voi osallistua henkilöitä ympäri maapallon, ei viestintä yleensä ole reaaliaikaista. Tämän vuoksi tärkein kommunikointiväline on sähköposti. Tavallisesti projekteilla on oma sähköpostilista, joka arkistoidaan. On myös tavallista, että kehittäjät, jotka ovat aktiivisina samaan aikaan vuorokaudesta, käyttävät IRC-palvelua kommunikointiin. Tällöin keskustelu on lähes reaaliaikaista. Myös IRC-keskustelut voidaan tallentaa, mutta tämä ei ole niin yleistä kuin postilistojen arkistointi.

10. Projekteilla ei yleensä ole minkäänlaisia virallisia suunnitelmia. Kokonaisuus etenee pienten muutosten johdosta evoluutiomaisesti. Varsinaista päämäärää ei tavallisesti ole määritelty, vaan tavoitteena on vain parantaa projektin nykytilaa jollain tavalla. Joissain tapauksissa aliprojekteilla voi olla jonkinlaisia suunnitelmia, mutta ne keskittyvät lähitulevaisuuteen kauaskantoisempien tavoitteiden sijaan.

11. Kehittäjien tiedot ja taidot vaihtelevat suuresti. Kenenkään ei tarvitse osata kaikkea, vaan jokainen voi keskittyä omien vahvuuksiensa käyttöön. Myös projektiin varsinaisesti kuulumattomat sivulliset voivat kantaa kortensa kekoon, esimerkiksi antamalla parannusehdotuksia tai oikolukemalla dokumentaatiota.

12. Vaikka suunnitelmia ei juurikaan ole, kehitettävän sovelluksen arkkitehtuuri muodostuu varsin usein erittäin modulaariseksi. Tämä johtuu suurelta osin käytännön syistä. Eri komponenttien väliset kytkökset pysyvät vähäisinä, kun eri aliprojektit tekevät eri komponentteja samanaikaisesti. Tämä mahdollistaa myös projektin hajautuksen, kun eri osat ovat riippumattomia toisistaan.

13. Projekteissa käytetään yleensä tunnettuja ja helposti saatavia työkaluja. Tämän vuoksi kuka tahansa voi hankkia itselleen tarvittavat ohjelmat ja liittyä projektiin. Työkalujen siirrettävyyden vuoksi kehitysympäristöäkään ei ole rajoitet-

tu, vaan eri kehittäjät voivat toimia hyvinkin erilaisissa ympäristöissä toisistaan riippumatta.

14. Kaikki projektiin liittyvä informaatio on vapaasti saatavilla. Dokumentaation lisäksi myös kehittäjien ja käyttäjien uutisryhmät, testitulokset ja virhetietokannat ovat yleensä julkisia. Tämä avoimuus mahdollistaa projektin seuraamisen myös sen ulkopuolelta. Osaltaan tämä on myös syynä siihen, että uudet kehittäjät pääsevät halutessaan helposti mukaan projektiin.

Kaikille piirteille on yhteistä se, että ne pyrkivät pitämään kehitysprojektin mahdollisimman avoimena. Vaikka jonkinlainen koordinaatio on välttämätöntä, perusoletuksena on, ettei kehittäjiä pitä rajoittaa turhaan, sillä he toimivat kyllä oikein, kunhan siihen annetaan mahdollisuus. Monille osallistujille ohjelmointi onkin luovaa taidetta, "Art of Computer Programming", johon he osallistuvat ilmaistakseen itseään, joten taiteilijoille on annettava työrauha. Käytäntö muistuttaa melkoisesti alkuperäisen hakkerikulttuurin toimintatapoja, mikä ei ole aivan sattumaa.

3.6 Yhteenveto

Sekä vapaiden ohjelmien että avoimen lähdekoodin kannattajien yhteisenä tavoitteena on saada vapaiden / avoimen lähdekoodin ohjelmien käyttö lisääntymään. Suurin ero näyttäytyy siinä, mikä painoarvo ohjelman lähdekoodin saatavuudelle annetaan.

Vapaiden ohjelmien liikkeen mukaan omisteiset ohjelmat muodostavat sosiaalisen ongelman ja ovat uhkana yhteiskunnalle. Ratkaisuksi tähän ongelmaan tarjotaan vapaita ohjelmia.

Avoimen lähdekoodin liike ei pidä omisteisia ohjelmia uhkana. Sen sijaan perinteisiä ohjelmistokehitysmenetelmiä käyttävien ohjelmien uskotaan hyötyvän avoimen lähdekoodin kehitysmenetelmien käyttöönotosta. Koska avoimen lähdekoodin ohjelmien on todettu olevan laadukkaita, ne pystyvät kilpailemaan kaupallisten ohjelmien kanssa teknisillä ominaisuuksillaan.

Bruce Perensin ja Eric S. Raymondin näkemysten ero on siinä, että Perensin alkuperäisenä tarkoituksena oli, että suljettujen ohjelmien vaihtoehdoksi valittaisiin vapaita ohjelmia, mikä oli Stallmaninkin tavoite. Raymond kuitenkin esitti, että vapaat ohjelmat eivät itsessään ole mitään tavoiteltavaa. Se, mitä kannattaa tukea, on hyväksi osoittautunut kehitysmenetelmä.

Ero Bruce Perensin ja Richard Stallmanin kantojen välillä on se, että Perensin mukaan vapaita ohjelmia ei tule levittää “väkisin”, sillä ne yleistyvät muutenkin; Stallmanin mukaan tämä on liian passiivista toimintaa.

Vapaiden ohjelmien kannattajat eivät kannata Eric S. Raymondin näkemystä, koska se jättää eettiset näkökohdat huomiotta. Avoimen lähdekoodin kannattajat taas katsovat, että eettisten kysymysten pohtiminen vaikeuttaisi liikkeen asemaa kilpailijoihin nähden.

Loppujen lopuksi erot ovat joko suuret tai pienet, näkökulmasta riippuen; alkuperäinen avoimen lähdekoodin liike eroaa vapaiden ohjelmien liikkeestä vain siinä, että se tuo asiat esiin maltillisemmin. Uudempi liike puolestaan edustaa aivan eri asiaa, kehitysmenetelmää, joka on vain hakkerikulttuurin sivutuote.

4 Avoimen lähdekoodin versionhallintaohjelmat

> *The essential feature of technological innovation is that it is an activity that is fraught with many uncertainties. This uncertainty, by which we mean an inability to predict the outcome of the search process, or to predetermine the most efficient path to some particular goal, has a very important implication: the activity cannot be planned.*
>
> — *Nathan Rosenberg*

Nyt kun versionhallinnan teoreettiset perusteet on käsitelty, voidaan siirtyä tarkastelemaan asioita toisesta näkökulmasta. Tässä luvussa käydään käsiksi varsinaiseen tutkimusongelmaan, joten pian katsotaan, minkälaisia versionhallintasovelluksia avoimen lähdekoodin puolelta löytyy ja kuinka ne ovat kehittyneet.

4.1 Tutkimusongelma ja tutkimusmenetelmä

Tässä tutkielman empiirisessä osassa analysoidaan avoimen lähdekoodin versionhallintaohjelmia ja niiden välisiä vertailuja *dialektisella* menetelmällä. On varsin selvää, että jokainen tiettyä ohjelmaa kehittävä yhteisö on laatinut omat ohjelmaesittelynsä ja testinsä siten, että ne tuovat esiin oman ohjelman parhaat puolet ja muiden heikkoudet. Näitä "puolueellisia" testituloksia yhdistelemällä on kuitenkin mahdollista saada todellisuutta vastaava käsitys, jossa otetaan huomioon kaikki tarpeelliset näkökulmat.

4.1.1 Tutkimusongelma

Palataan aluksi luvussa 1 mainittuun "hiljaiseen tietoon". Tällä käsitteellä tarkoitetaan filosofi Michael Polanyin esittämää sellaista omattua tietoa, joka vaikuttaa toimintaan, mutta jonka sisältöä ei pystytä yksiselitteisesti esittämään. Hyviä esimerkkejä tällaisesta tietämisestä ovat esimerkiksi henkilön tunnistaminen kasvojen perusteella tai vaikkapa polkupyörällä ajaminen: kukapa osaisi sanoa täsmällisesti, kuinka ne tapahtuvat. [22, 67]

1990-luvun puolivälissä japanilaiset Ikujiro Nonaka ja Hirotaka Takeuchi jul-

kaisivat kirjan “The Knowledge-Creating Company”, jossa he laajensivat Polanyin idean käsittämään yksilöiden lisäksi myös yritykset. Heidän mukaansa yrityksillä oleva hiljainen tieto (tacit knowledge) on keskeinen voimavara, joka täsmälliseksi tiedoksi (explicit knowledge) muutettuna voi antaa merkittävän kilpailuedun. Tätä käsitystä on kuitenkin kritisoitu mm. siksi, että Polanyin alkuperäisen mallin mukaan hiljaista tietoa ei ole mahdollista esittää – “We can know more than we can tell”. [22, 67]

Tämän tutkielman taustaksi on asetettu kaksi hypoteesia: ensimmäisen mukaan myös avoimen lähdekoodin projekteihin kätkeytyy hiljaista tietoa, ja toisen mukaan tämän tiedon olemassaolo voidaan päätellä projektien yhteydessä esiintyvistä ideoista.

Ensimmäinen hypoteesi voidaan katsoa paikkansapitäväksi jo sillä perusteella, että Polanyin mukaan kaikki tieto perustuu hiljaiseen tietoon, joten eksplisiittisen tiedon olemassaolo on osoitus hiljaisen tiedon olemassaolosta. [67]

Toinen hypoteesi onkin hieman vaativampi. Sen pätevyys voidaan osoittaa nostamalla esiin sellaista tietoa tai sellaisia ideoita, joista ei ole yleisesti saatavilla eksplisiittistä tietoa, jolloin niiden on perustuttava hiljaiseen tietoon.

Tässä tutkielman käytännöllisessä osassa keskitytään tämän toisen hypoteesin selvittelyyn. Käytännössä tämä tehdään niin, että tässä osassa tutustutaan erilaisiin avoimen lähdekoodin versionhallintaohjelmiin, varsinkin niiden erityisominaisuuksiin. Tarkoituksena on omia havaintoja tekemällä sekä valmiita testejä ja vertailuja tutkimalla selvittää, mitä omaperäistä eri ohjelmissa on, tai mitä sellaisia ongelmia niissä on ratkaistu, joihin tieteellisessä tutkimuksessa ei ole vielä kajottu.

4.1.2 Tutkimusmenetelmä

Yksi tapa löytää eri versionhallintaohjelmissa olevia erityispiirteitä on yksinkertaisesti vertailla niitä toisiinsa, jolloin niissä olevat yhteiset piirteet löytyvät. Jos sovelluksista näiden yhtäläisyyksien lisäksi sitten löytyy vielä sellaisia ominaisuuksia, joita ei ole aiemmin käsitelty, ne ovat juuri niitä erityisominaisuuksia, joita kannattaa tarkastella lähemmin. Tietenkin on myös mahdollista, että jokin alun perin yhden sovelluksen käyttöön ottama innovaatio on levinnyt yleiseen käyttöön, mutta tällöin kyseessä ei enää olekaan mikään tuntematon asia, vaan se on luultavimmin jo perin pohjin tutkittu.

Sovellusten vertailua varten tarvitaan perusteelliset tiedot käsiteltävistä ohjelmista. Nykyisenä Internet-aikakautena tietojen saatavuuden kannalta tässä ei ole

mitään ongelmaa, mutta määrän ja laadun suhteen asia on toinen: aihetta käsitteleviä dokumentteja, sähköpostiviestejä, wiki-sivuja ja blogeja on niin paljon, että niiden kaikkien läpikäynti on mahdotonta.

Tässä tutkielmassa käytettyä aineistoa on kerätty pääasiassa eri sovellusten www-sivustoilta ja niiden käyttämien sähköpostilistojen viestiarkistoista. Rajaus tehtiin sillä perusteella, että www-sivuja katsottiin käytettävän sovellusten esittelyyn ja "markkinointiin", ja sähköpostilistoja niiden todellisen kehityksen vaatimaan kommunikointiin. Näitä tutkimalla voidaan saada kuva siitä, mitä sovelluksista halutaan julkisesti sanoa ja kuinka niiden kehitys todellisuudessa etenee.

4.1.3 Tutkimusmenetelmän arviointia

Tutkimuksessa tehdään aineistosta päätelmiä Pengin [50] esittämän kiinalaisen dialektiikan mukaisesti. Koska menetelmä on vähän tunnettu eikä sitä ole aiemmin sovellettu (ainakaan nimeltä mainiten) tietotekniikka-alan opinnäytteissa, sen taustoja on hyvä esitellä lyhyesti.

Antiikin ajan filosofi Sokrateelle dialektiikka tarkoitti menetelmää, jossa eri näkemyksiä jostain asiasta tarkasteltiin useasta eri näkökulmasta. Eri näkemysten avulla keskustelua pyrittiin johtamaan siten, että jotkut näkemykset johtivat ristiriitaan itsensä kanssa. Näin menettelemällä vain parhaat (eli oikeimmat) näkemykset jäivät jäljelle.

Länsimaissa dialektiikka tunnetaan erityisesti käsityksenä, missä *teesi* ja sen vastakohta *antiteesi* johtavat *synteesiin*. Esimerkiksi Karl Marx antoi ristiriidoille tärkeän merkityksen kehityksen eteenpäinviejänä.

Tämän näkemyksen mukaan edistystä tapahtuu silloin, kun kaksi tai useampi keskenään ristiriitaista näkemystä asetetaan vastakkain. Ristiriita ratkeaa joko jonkun näkemyksen voittoon tai siihen, että eri näkemysten välille rakentuu jonkinlainen kompromissi. Voittanut näkemys on vallassa jonkin aikaa, kunnes uusia haastajia uusine ristiriitoineen ilmaantuu. Sama kierre toistuu uudelleen ja uudelleen.

Perimmäisenä ajatuksena tässä näkemyksessä on, että ristiriitatilanteissa jonkin näkemyksen on oltava oikea ja muiden vääriä. Tämä juontaa juurensa antiikinaikaiseen logiikkaan, jossa asiat joko ovat tai eivät ole, mutta eivät voi olla kumpaakin samanaikaisesti.

Kiinalainen dialektiikka ei pohjaudu länsimaissa vallitsevaan joko-tai-ajatteluun. Siinä lähtökohtana on näkemys, että jokainen ristiriitainenkin näkemys voi omalta osaltaan olla oikeassa. Tämä länsimaalaiselle ajattelulle vaikeasti nieltävä ajatus on

helpompi käsittää, kun sen asettaa oikeaan kontekstiin.

Kiinalaisen ajattelutavan mukaan koko olemassaolo on jatkuvan muutoksen tilassa. Asiat voivat muuttua hetkessä, vaikka ne näyttäisivätkin olevan pysyviä. Samalla tavoin asia, joka eilen oli mahdoton, onkin tänään täysin mahdollinen. Siten eilisen näkemyksen ja tämänpäiväisen näkemyksen välillä ei todellisuudessa olekaan ristiriitaa, sillä ne vain kuvaavat oman aikansa todellisuutta.

Tämä lähestymistapa sopii oikein hyvin versionhallintaohjelmien tutkimiseen, sillä etenkin avoimen lähdekoodin projekteissa muutokset voivat tapahtua jopa yhdessä yössä. Muutosten nopeus johtaa siihen, että viikon vanha arvostelu voikin olla jo paikkansapitämätön. Kun tutkittava ajanjakso on vuosien mittainen, on selvää, että vastaan tulee monia ristiriitaisia näkemyksiä, jotka ovat kuitenkin omassa kontekstissaan oikeita. Näiden näkemysten arviointiin kiinalainen dialektiikka sopii paremmin kuin hyvin.

Koska tutkimustulosten on oltava mahdollisimman luotettavia, niiden oikeellisuuteen vaikuttaviin seikkoihin on kiinnitettävä riittävästi huomiota. Erityisen tärkeää on pohtia tutkittavaan aineistoon sisältyviä virhemahdollisuuksia ja säilyttää riittävä lähdekriittisyys. On esimerkiksi erittäin todennäköistä, että eri versionhallintaohjelmia kehittävien yhteisöjen itse laatimat ja julkaisemat dokumentit ja testitulokset esittävät oman sovelluksen mahdollisimman positiivisessa valossa. Tästä syystä aineiston on oltava niin monipuolista, että mukaan saadaan mahtumaan myös “kilpailijoiden” tekemät arvioinnit, jotka omalta osaltaan voivat tasapainottaa tulosta.

Ristiriitaisten näkemysten yhteensovittaminen voi olla ongelmallista, mutta se on myös hyvä keino saada todellinen sisältö esiin mainoslauseiden takaa.

Kehitykseen käytettäviä sähköpostilistoja voidaan pitää julkisia www-sivustoja luotettavampina lähteinä, koska ne sisältävät yleensä realistisempia arvioita ohjelmien toiminnasta. Usein postilistalla esitetyt vertailut myös poikkeavat dokumentaatiossa esitetyistä siinä, että niissä oma ohjelma saattaa menestyä yllättävän huonosti. Tämä johtaa yleensä ohjelman huonon menestyksen arviointiin ja mahdollisiin korjaustoimenpiteisiin.

4.1.4 Sovellusten valinta

Avoimen lähdekoodin versionhallintaohjelmien suuren määrän vuoksi niiden kaikkien löytäminen tai tarkastelu on lähestulkoon mahdotonta. Tämän vuoksi arvioitavaksi pyrittiin saamaan sellainen ohjelmajoukko, joka edustaa ohjelma-aluetta mah-

dollisimman monipuolisesti. Aluksi tätä tarkoitusta varten laadittiin lista kaikista niistä sovelluksista, joita pystyttiin mm. Wikipedian, hakukoneiden tai yleisen kokemuksen avulla löytämään. Tämän jälkeen joukosta karsittiin pois kaikki sellaiset ohjelmat, joiden ei katsottu olevan tutkielman kannalta merkityksellisiä. Tärkeimmät syyt karsiutumiselle olivat nämä:

- Sovelluksesta ei löytynyt mainintoja muiden versionhallintaohjelmien kehityslistoilta tai www-sivustoilta.
- Sovellus ei ollut itsenäinen, vaan jonkin kehitysympäristön osa.
- Sovelluksesta ei pikaisessa tarkastelussa löytynyt mitään omintakeisia ratkaisuja.
- Sovellus oli liian samankaltainen jonkin toisen tarkasteltavan ohjelman kanssa.
- Sovelluksen tulevaisuus ei vaikuttanut lupaavalta.

Karsintakriteereitä sovellettiin järjestyksessä harkinnan mukaan. Esimerkiksi jotkut hyvin tunnetut sovellukset pääsivät mukaan jo ensimmäisen kriteerin perusteella, mutta vähemmänkin tunnetut saivat uuden mahdollisuuden. Ennen hylkäystä niistä etsittiin omintakeisia ratkaisuja, joiden olemassaolo varmisti sovelluksen pääsyn lähempään tarkasteluun. Valittujen sovellusten joukosta karsittiin lopussa pois vielä toistensa kloonit sekä sellaiset sovellukset, joiden tulevaisuus ei näyttänyt lupaavalta. Käytännössä karsinnassa tippuivat seuraavat sovellukset:

Archipel http://wiki.type-z.org/index.php/Projects/Archipel

ASVCS (A Simple Version Control System) http://asvcs.com/

Barch (Binary Arch), http://www.robertcollins.net/barch-0.0.6-DEVEL.tar.bz2

/BriefCase http://www.applied-cs-inc.com/bcintro.html

CBE (Code Building Environment), http://cbe.sourceforge.net/

EVA http://initd.org/pub/software/eva/

GAT http://www.scylla-charybdis.com/tool.php?tool=gat

gennf http://gennf.berlios.de/

JEDI VCS http://jedivcs.sourceforge.net/

Katie http://www.netcraft.com.au/geoffrey/katie/

OpenCM http://www.opencm.org/

OpenCVS http://www.opencvs.org/

ShapeTools http://swt.cs.tu-berlin.de/~shape/

SiVeCo (Simple Version Control), http://kisocd.sourceforge.net/page11021844.htm

Karsintakriteerit olivat varsin subjektiiviset, joten pudotettujen sovellusten joukossa saattaa olla myös sellaisia ohjelmia, joihin olisi kannattanut perehtyä hieman tarkemmin. Lähempään tarkasteluun päässeiden sovellusten joukkoa voidaankin oikeutetusti kritisoida, mutta tämän tutkielman osalta se täyttää tehtävänsä nykymuodossaankin.

4.1.5 Sähköpostiarkistojen käyttö

Avoimen lähdekoodin periaatteiden mukainen kehitys on julkista, mikä antaa hyvän mahdollisuuden seurata sovellusten edistymistä ja todellista kehitystapaa. Tässä tutkielmassa julkisuutta hyödynnettiin käymällä läpi suuri määrä eri versionhallintaohjelmien kehityslistoille lähetettyjä sähköpostiviestejä. Käytettyjen sähköpostiarkistojen osoitteet kerrotaan kunkin ohjelman esittelyn yhteydessä, ja läpikäytyjen viestien arvioitu määrä selviää taulukosta 4.1.

Taulukossa esitetyt viestimäärät ovat arvioita, koska joidenkin ohjelmien sähköpostilistat sisälsivät niin paljon roskapostia, että asiallisten viestien todellinen määrä piti laskea virhealttiisti käsityönä. Muutamassa tapauksessa viestit myös jakautuivat usealle eri listalle, tai listoja ei ollut lainkaan. Kaiken lisäksi viestimäärät vaihtelivat eri arkistojen välillä, mikä saattaa johtua erilaisesta roskapostisuodatuksesta. Näiden seikkojen vuoksi kuvatut viestimäärät ovat lähinnä suuntaa-antavia, mutta niistä voi saada oikeansuuntaisen käsityksen eri sovellusten kehityksen aktiivisuudesta.

4.2 Tutkimuksen taustaa

Edeltävissä luvuissa kuvattiin versionhallinnan kehittyminen sen alkuajoista 1990-luvun loppupuolelle. Tieteellisen tutkimuksen mielenkiinto sitä kohtaan näyttää lai-

	2005				2006				2007		
	1	**2**	**3**	**4**	**1**	**2**	**3**	**4**	**1**	**2**	**3**
Aegis	4	4	0	0	1	0	0	0	98	58	34
Arch	880	580	497	633	431	292	67	44	28	14	44
ArX	65	84	61	36	0	2	20	4	0	6	4
Bazaar	41	1614	1595	3362	3878	3738	3515	2954	3383	3058	4805
Codeville	—	121	47	15	6	8	6	0	5	2	2
Darcs	664	1095	608	475	484	153	382	437	267	357	459
GIT	—	5878	4092	4549	4048	4787	5150	7560	7850	7716	9036
Mercurial	—	1725	3023	1330	1350	1628	2005	950	660	884	1311
Monotone	678	1327	905	692	771	835	1379	1122	1042	777	1057
PRCS	8	0	0	0	0	0	0	0	0	0	0
RCS	5	6	6	5	2	4	0	2	5	2	1
SourceJammer	136	27	12	4	0	15	0	0	—	—	—
Subversion	3717	4454	4108	4090	4125	2864	3119	2138	2982	2605	2288
Superversion	120	22	43	36	37	7	11	2	1	5	0
SVK	172	226	173	205	154	113	251	263	365	168	91
Vesta	140	118	46	51	19	34	40	45	136	98	106

Taulukko 4.1: Sähköpostiviestien määrä neljännesvuosittain.

menneen tämän ajanjakson loppupuolella, ja tutkimuksen painopiste onkin enenevissä määrin siirtynyt konfiguraationhallinnan muiden osa-alueiden puolelle. Yhtenä syynä tähän on se, että versionhallintaa pidetään nykyään niin hyvin tunnettuna, että sen edistymisen katsotaan nyt riippuvan saatujen tutkimustulosten soveltamisesta käytäntöön. Frühauf ja Zeller muistuttavatkin, että moniin ongelmiin on jo olemassa tunnettuja ratkaisuja, mutta niitä ei vielä hyödynnetä kaikissa sovelluksissa. [19]

Tieteen näkökulmasta versionhallinnan tärkeimpiä asioita ovat siihen liittyvät käsitteet, teoriat ja algoritmit. Versionhallintaohjelman käyttäjälle tärkeämpää on kuitenkin se, miten sovellus oikeasti toimii. Näiden näkökantojen välille muodostuva ristiriita aiheutuu usein siitä, ettei tieteellinen mielenkiinto aina kohdistu niihin käytännön ongelmiin, joihin tarvittaisiin ratkaisua.

Linus Torvaldsin mukaan käytännössä vastaantulevat ongelmat ovat selvästi haasteellisempia kuin ne, joita yleensä tutkitaan[1]:

[1] http://www.gelato.unsw.edu.au/archives/git/0610/29833.html (viitattu 25.1.2007).

> The stuff you can think through (and argue about) tends to be the easy stuff. Exactly because you _can_ think about it abstractly.
>
> The stuff that is actually really hard and time-consuming is the stuff that you find out in practice, and you have to iterate on.

Linus mainitsee esimerkkinä vaikeista ongelmista Linux-ytimen laitteistoriippuvat osat. Hän toteaa laitehallinnan olevan huomattavasti muistinhallintaa ja prosessien vuorottamista (engl. scheduling) vaikeampaa. Tämä vaikeus ei hänen mielestään kuitenkaan näy tieteellisissä julkaisuissa, sillä hän arvioi niiden osuuden kaikista julkaisuista olevan alle prosentin.

Sama suhtautuminen käytännön ongelmiin näkyy myös versionhallinnan puolella. Linus antaa seuraavan esimerkin käytännön ongelmasta:

> [...] I suspect that in git just the CVS importer has gotten _way_ more attention than merging ever got. Importing from CVS is simply a much harder problem in practice [...] It's hard to "think" about, because a lot of the problems with importing from CVS are literally all about the details and the nasty crud. I really think "merging" is _way_ easier.

Tästä voitaneen päätellä, että esimerkiksi tietojen siirto versionhallintajärjestelmästä toiseen on lähes tutkimaton ongelma-alue. Tätä päätelmää tukee se, ettei aiheesta juurikaan löydy käytännön ongelmiin vastaavaa materiaalia, mutta erilaisia yhdistämismenetelmiä käsitellään vaikka kuinka paljon – siitä huolimatta, että yksinkertainen 3-way-merge on jo osoittanut toimivuutensa. Yhtenä syynä tähän tilanteeseen voidaan pitää sitä, että tieteessä asioita halutaan käsitellä laajoina kokonaisuuksina ja pitkällä aikajänteellä. Yksittäisiä sovelluksia tulee ja menee, joten niihin liittyvä tietämys muuttuu varsin nopeasti arvottomaksi.

Vaikka yksittäisten sovellusten merkitys usein jääkin melko vähäiseksi, niissä olevat ideat voivat tehdä niistä joskus hyvinkin pitkäikäisiä. Esimerkiksi taulukossa 4.2 esitettävien Estublierin ym. [16] kokoamien versionhallinnan ja konfiguraationhallinnan merkkitapahtumien joukosta löytyvät *Diff* ja *Make* ovat jossakin muodossa käytössä vielä nykyäänkin.

Vaikka taulukossa mainitut versionhallintaan liittyvät seikat ovatkin hyvin tunnettuja, on muistettava, että ala kehittyy edelleen. Etenkin avoimen lähdekoodin projekteissa asiat voivat jo muutamassa vuodessa muuttua melkoisesti, joten niitä koskevat tiedot on muistettava päivittää riittävän usein. Muutosten nopeus näkyy myös versionhallintaohjelmissa, sillä esimerkiksi tutkielmassa käsiteltävistä sovelluksista jopa 12 on tehty vuosien 2003–2005 aikana, kuten taulukosta 4.3 voi nähdä.

	Academic Research	**Industrial Research**	**Industrial Product**
1972		SCCS (Bell Labs)	
1976		Diff (Bell Labs)	
1977		Make (Bell Labs)	
1980	Variants, RCS (Purdue University)		
1980		Change-sets (Xerox Parc)	
1982	Merging, and/or graph (Purdue University)		
1983		Change-sets (Aide-de-Camp)	
1984	Selection (Grenoble University)		
1985		System model (DSEE)	
1988	First International SCM workshop		
1988	Process support (Grenoble University)		
1988	NSE Workspaces (Carnegie Mellon University; Sun)		
1990		3DFS, nDFS virtual file system (Bell Labs)	
1994		Virtual file system (ClearCase)	
1994		MultiSite (ClearCase)	
1996		Activity-oriented SCM (Asgard, Bell Core)	
2000	WebDAV/DeltaV (University of California, Irvine, Microsoft, ClearCase, ...)		

Taulukko 4.2: Versionhallinnan merkkitapahtumia.

On mielenkiintoista huomata, kuinka avoimen lähdekoodin versionhallintaohjelmien määrä 2000-luvulla on selvästi kasvanut. Vuosina 2003 ja 2005 uusia sovelluksia tehtiin erityisen innokkaasti, mikä saattaa selittyä BitKeeperin käyttöönoton ja myöhemmän BitKeeper-kriisin (2.15.1, s. 39) saamalla julkisuudella. Merkittävää etenkin näissä uusimmissa sovelluksissa on se, että niissä kaikissa on pyritty vastaamaan Midhan jo vuonna 1997 esittämiin uusiin haasteisiin [45]:

- Kehitys muuttuu hajautetuksi.
- Kehityksestä tulee rinnakkaista ja samanaikaista.
- Sovelluksen on toimittava useissa käyttöjärjestelmissä.
- Sovelluksen on tuettava komponenttipohjaista kehitystä.

vuosi	ohjelmat
1972	SCCS
1982	RCS
1985	CVS
1991	Aegis
1993	Vesta
1997	CSSC, PRCS
2000	Subversion
2001	SourceJammer
2002	Arch, DCVS, Meta-CVS
2003	ArX, Codeville, Darcs, Monotone, Superversion, SVK
2004	Pastwatch
2005	Bazaar, Bky, FastCST, GIT, Mercurial

Taulukko 4.3: Sovellusten julkaisuvuodet.

- Sovelluksen on tuettava Internet-pohjaista kehitystä.

Nykyisille versionhallintaohjelmille onkin asetettu paljon vaatimuksia, joita varten on ollut pakko keksiä erilaisia sopeutumiskeinoja. Vaikka nämä ratkaisut voivat olla hyvinkin epätieteellisiä, ne ovat kuitenkin riittäviä jokapäiväiseen käyttöön. Tätä taustaa vasten onkin mielenkiintoista katsoa, milloin jokin versionhallintaohjelma erottuu joukosta niin omintakeisilla ja kestävillä ratkaisuilla, että se onnistuu herättämään myös tieteellisen mielenkiinnon.

Seuraavissa aliluvuissa selvitellään sitä, kuinka versionhallinta on kehittynyt sen jälkeen, kun tieteellinen mielenkiinto sitä kohtaan on vähentynyt. Hypoteesina on, että moni avoimen lähdekoodin piirissä syntynyt sovellus on tuonut versionhallinnan alalle mukanaan jotain sellaista uutta, jolla on annettavaa myös tieteelliselle tutkimukselle.

4.3 Tutkimuksen toteutus

Tutkimus alkoi laajalla kirjallisuuskatsauksella, jonka tarkoituksena oli selvittää avoimen lähdekoodin versionhallintaan liittyvä nykytietämys. Tämän jälkeen lähempään tarkasteluun valittiin kirjallisuudesta saatujen viitteiden ja myöhemmin esiteltävien muiden lähteiden perusteella joukko erilaisia sovelluksia, joista pyrittiin mm.

vertailujen avulla löytämään mahdollisia innovaatioita.

Aineiston läpikäynnissä lähdettiin siitä, että jokaista tutkittavaa sovellusta tarkasteltiin sekä yksittäistapauksena että osana laajempaa avoimen lähdekoodin versionhallintaohjelmien joukkoa. Tämä lähestymistapa auttoi erillisten sovellusten ja koko sovellusalueen suhteiden hahmottamista, mutta myös nosti esille ns. emergenttejä ominaisuuksia, joita yksittäisten sovellusten yhteydessä ei olisi voinut havaita.

Seuraavissa aliluvuissa tutustutaan joukkoon erilaisia avoimen lähdekoodin versionhallintaohjelmia. Jokaisesta ohjelmasta annetaan perustiedot, kuten käytetty lisenssi ja www-sivuston osoite, ja esitetään joitakin sovellukselle tyypillisiä piirteitä, joiden perusteella sen toiminnallisuudesta saadaan oikeanlainen kuva.

Jokaista sovellusta kommentoidaan jonkin verran, mutta varsinaiseen arviointiin ei vielä tässä vaiheessa lähdetä. Mikäli ohjelma sisältää joitakin muista sovelluksista poikkeavia innovatiivisia toimintoja, ne mainitaan tässä lyhyesti. Erityisominaisuuksia käsitellään tarkemmin ohjelmaesittelyiden jälkeen kohdasta 5.4 (sivu 195) alkaen.

Joidenkin sovellusten kohdalla esitetään myös niiden kehitykseen liittyvää todellista historiaa, jota on selvitelty sähköpostiarkistojen perusteella. Tämän tarkoituksena on antaa viitteitä siitä, kuinka eri sovellusten kehittäjien välinen yhteistyö on toiminut ja kuinka ideoita on vaihdettu puolin ja toisin.

Sovelluksiin tutustuminen alkoi vierailuilla niiden www-sivustoille ja selvittämällä niihin liittyvän muun aineiston, kuten wiki-sivustojen, blogien ja sähköpostilistojen, olemassaolo. Sivustoilla esitetyt tiedot kerättiin talteen aivan kritiikittömästi, eli niiden oikeellisuutta ei vielä tässä vaiheessa kyseenalaistettu.

Aineiston keräämisen jälkeen alettiin tarkastella sen paikkansapitävyyttä. Koska läpikäytyjen sivustojen ylläpitäjiä ei ollut aihetta pitää epärehellisinä, katsottiin tarpeelliseksi tarkistaa vain sellaiset tiedot, jotka voivat vanhentua. Tämän vuoksi esimerkiksi sivustoilla ilmoitetut lisenssit tarkastettiin, mutta mahdollisesti esitettyä sovelluksen historiaa ei selvitetty tarkemmin, koska sitä ei ollut syytä epäillä.

Sovellusten vertailussa hyödynnettiin omien testien ja havaintojen lisäksi myös kehityslistoilla esitettyjä vertailuja ja kehittäjien blogeja, sekä erityisesti seuraavia sivustoja:

- http://better-scm.berlios.de/comparison/comparison.html
- http://linuxmafia.com/faq/Apps/scm.html

- http://www.dwheeler.com/essays/scm.html
- http://www.gnuarch.org/gnuarchwiki/SubVersionAndCvsComparison
- http://www.vestasys.org/doc/comparison.html

Tietojen oikeellisuus pyrittiin varmistamaan vertaamalla useiden eri lähteiden tietoja toisiinsa. Monissa tapauksissa tämä paljasti yksittäisiä virheitä, mutta aina yhteisymmärrystä ei näyttänyt löytyvän. Tällöin asiat tarkastettiin lähdekooditasolla, eli esimerkiksi lisenssikiistoissa ratkaisi lopulta se, mikä lisenssi sovelluksen uusimpaan lähdekoodiin oli merkitty.

4.4 SCCS ja CSSC

SCCS (Source Code Control System) [57] on Marc J. Rochkindin kehittämä ensimmäinen varsinainen versionhallintajärjestelmä. Se toteutettiin alun perin SNOBOL4-kielellä IBM 370 -järjestelmään, mutta laajemman kohderyhmän tavoittamiseksi se siirrettin myöhemmin PDP-11-alustalle Unix-järjestelmään. SCCS ei ollut vapaa ohjelma, mutta vuosina 1975–1985 se oli käytännössä ainoa järkevä vaihtoehto versionhallintaan.

4.4.1 Ominaisuudet

Koska SCCS oli ensimmäinen versionhallintaohjelma, se pääsi käytännössä määrittelemään versionhallintaohjelmilta vaadittavat perusominaisuudet. Seuraavassa esitetään näiden ominaisuuksien lisäksi se, kuinka SCCS ne itse toteutti.

Tallennus (engl. storage) Tiedoston kaikki versiot tallennetaan siten, että jokaiseen versioon pääsee myöhemmin käsiksi. SCCS:n käyttämässä weave-muodossa (s. 14) kaikki tiedoston versiot tallennetaan samaan tiedostoon limittäin niin, ettei eri versioihin kuuluvia samankaltaisia osia tarvitse tallentaa moneen kertaan.

Suojaus (engl. protection) Kehittäjän pääsyä eri versioihin voidaan rajoittaa. SCCS lukitsee tiedoston muokkauksen ajaksi, joten sitä voi muokata vain järjestelmän luvalla.

Identifiointi (engl. identification) Jokaiseen versioon liitetään tunnistetiedot, joiden avulla niihin voidaan viitata. SCCS käyttää versioiden identifiointiin versionumeroita.

Dokumentointi (engl. documentation) Jokaiseen tehtyyn muutokseen liitetään tiedot muutoksen tekijästä, ajankohdasta ja muutoksen syystä.

SCCS tallentaa tiedot etenevinä deltoina. Perinteisten deltojen lisäksi SCCS käyttää kahta erikoista deltatyyppiä, valinnaisia deltoja ja lisäys- ja poistodeltoja.

Valinnaisiin deltoihin liittyy ns. *optiokirjain* eli yksittäinen merkki, jonka avulla määritellään, otetaanko kyseinen delta mukaan versioon vai ei. Mekanismin tarkoituksena on mahdollistaa asiakaskohtaisten muutosten säilyttäminen. Kun tiedoston jotain versiota otetaan ulos versionhallinnasta, ohjelmalle voidaan antaa parametrinä haluttu optiokirjain. Tällöin kaikki kyseisen kirjaimen sisältämät optionaaliset deltat otetaan mukaan.

Lisäys- ja poistodeltat ovat deltoja, jotka pakottavat muiden deltojen mukaanoton tai poisjätön. Yksinkertaisimmillaan poistodeltaa voidaan käyttää virheellisen deltan ohittamiseen. Lisäysdeltan avulla optionaalinen delta voidaan pakottaa mukaanotettavaksi. Lisäys- ja poistodeltat voivat olla myös optionaalisia deltoja.

4.4.2 Toiminta

SCCS tallentaa versiotiedot työhakemiston `SCCS`-nimiseen alihakemistoon weave-muodossa[2].

SCCS käsittelee yksittäisiä tiedostoja. Uusi tiedosto laitetaan versionhallintaan komennolla `sccs create main.c`. Komento luo uuden versiotiedoston, siirtää alkuperäisen tiedoston eri nimelle ja ottaa ensimmäisen version ulos versionhallinnasta kirjoitussuojattuna.

Tiedoston muutos aloitetaan ottamalla se ulos tietovarastosta komennolla `sccs get main.c`. Komento merkitsee tiedoston lukituksi ja antaa käyttöön muokattavan version.

Muutosten jälkeen uusi delta luodaan komennolla `sccs delta main.c`. Komennolle voidaan antaa selostus, miksi muutos tehtiin. Tämän jälkeen komento tallentaa muutoksen ja poistaa muutetun tiedoston.

[2] Tämän aliluvun esimerkit on testattu GNU CSSC:n versiolla 1.0.1, joten kaikkien yksityiskohtien vastaavuutta alkuperäisessä SCCS:ssä ei voida taata.

Eri versioihin tehtyjä muutoksia voidaan katsoa komennolla `sccs print main.c`. Komento näyttää kuhunkin muutokseen liittyvän selosteen, muutoksen tekijän ja muutosajankohdan.

4.4.3 SCCS:n merkitys versionhallinnalle

SCCS oli radikaali uudistus, joka käytännössä aloitti koko versionhallintaohjelmien historian. Sen kehittäjä Marc J. Rochkind kertoo asiasta näin [57]:

> Because SCCS represented such a radical departure from conventional methods for controlling source code, it became clear when we began development of it (in late 1972) that a paper specification would not be sufficient to "sell" the system to the software projects for which it was intended; we would have to have a working prototype."

Jotakin SCCS:n merkityksestä kertoo jo se, että sen komentorivikäyttöliittymä on hyväksytty standardiin [34]. Tämän johdosta jokainen POSIX[3]-yhteensopiva järjestelmä tukee SCCS:n komentoja ainakin nimellisesti.

4.4.4 Lisenssi

Alun perin SCCS tuli Unixin mukana ja vaati siten kaupallisen Unix-lisenssin hankkimisen. Vuoden 2006 lopulla se kuitenkin julkaistiin[4] OpenSolaris-projektin[5] alaisuudessa, joten nykyisin se on käytettävissä avoimen lähdekoodin CDDL-lisenssin (Common Development and Distribution License) version 1.0 ehdoilla.

Koska SCCS:n merkitys versionhallinnalle oli niin suuri, siitä tehtiin ennen sen lähdekoodin vapautusta GPL-lisenssin alainen klooniversio, CSSC (Compatibly Stupid Source Control)[6], jonka tarkoituksena oli olla täydellinen kopio SCCS:stä, mahdolliset virheet mukaanlukien. Tavoitteestaan johtuen CSSC ei tarjoa mitään innovatiivista, mikä onkin tässä tapauksessa hyvä asia. Koska tavoitteena ei ole saavuttaa suurta käyttäjäkuntaa, on perusteltua pitäytyä vain sellaisten toimintojen tukemisessa, mitä tarvitaan tavoitteen saavuttamiseksi.

[3] Portable Operating System Interface

[4] http://mail.opensolaris.org/pipermail/opensolaris-announce/2006-December/001418.html (viitattu 19.9.2007).

[5] http://www.opensolaris.org (viitattu 19.9.2007).

[6] http://cssc.sourceforge.net/ (viitattu 1.8.2007),
http://sourceforge.net/mailarchive/forum.php?forum_name=cssc-devel.

4.4.5 Nykytila

SCCS oli Unix-järjestelmissä käytössä useita vuosia, ennenkuin RCS syrjäytti sen. Se toimii vieläkin riittävän hyvin pienillä projekteilla, mutta sen käyttöä uusissa projekteissa ei enää suositella. CSSC:n tarkoituksena onkin ollut tarjota keino saada SCCS-järjestelmässä olevat tiedot ulos, jotta ne voisi siirtää jonkin muun versionhallintaohjelman alaisuuteen. Lähdekoodin vapauttamisen jälkeen sillekään ei ole enää käyttöä, joten on hyvin todennäköistä, että nämä sovellukset jäävät piakkoin historiaan, jossa niillä on oma paikkansa.

4.5 RCS

RCS (Revision Control System) [65][7][8] on Walter F. Tichyn 1980-luvun alussa Purduen yliopistossa kehittämä versionhallintajärjestelmä. Valmistuessaan se sisälsi useita parannuksia silloin käytössä olleeseen SCCS-versionhallintaohjelmaan nähden, kuten helppokäyttöisemmän käyttöliittymän ja nopeamman vanhojen versioiden käsittelyn.

Alun perin RCS oli tarkoitettu lähinnä ohjelmistokehittäjien työkaluksi, mutta varsin pian sille ilmaantui myös muita käyttötapoja. Esimerkiksi eri järjestelmien ylläpitäjät alkoivat tallentaa siihen muokkaamiaan konfiguraatiotiedostoja, koska se mahdollisti mahdollisen virheen sattuessa palaamiseen aiemmin toimineeseen versioon. Yksi RCS:n eduista SCCS:ään nähden olikin juuri vanhojen versioiden käsittelyn nopeus. Tämä saatiin aikaan aiemmin esitettyjä takautuvia deltoja käyttäen, eli uusin versio oli käytettävissä kokonaisena ja aiempiin voitiin palata tallennettuja deltoja käyttämällä.

Vuonna 1991 RCS lisensoitiin GPL:n (GNU General Public License) alle ja adoptoitiin GNU-paketiksi, joten se on avoin (ja vapaa) ohjelma. Sen nykyinen ylläpitäjä on Paul Eggert.

4.5.1 Ominaisuudet

RCS tarjoaa tuen tekstitiedostojen, kuten lähdekoodin ja dokumentaation, eri versioiden tallentamiselle ja hakemiselle tietovarastosta, sekä lokitietojen hallitsemiselle ja eri versioiden identifioimiselle. Näiden lisäksi RCS tukee myös yksinkertaista

[7] http://www.cs.purdue.edu/homes/trinkle/RCS/ (viitattu 1.8.2007).
[8] http://www.gnu.org/software/rcs/rcs.html (viitattu 1.8.2007).

konfiguraationhallintaa.

RCS käsittelee vain yksittäisiä tiedostoja, joten se ei mahdollista atomisten muutosjoukkojen käyttöä. Yksittäisten tiedostojen käsittely on kuitenkin tehokasta, ja niiden suhteen tuetaan jopa eri kehityshaarojen käyttöä. Tiedostojen versiot tunnistetaan moniosaisella versionumerolla, joka on muotoa $x.y.\ldots$. Tämän mukaisesti esimerkiksi versionumerolla 1.2.4.1 tarkoitetaan tiedoston tilaa versiosta 1.2 alkaneen neljännen kehityshaaran alussa.

RCS tallentaa muutokset käänteisten deltojen avulla, joten uusimpien versioiden käyttö on nopeaa. Deltojen muodostuksessa käytetään Tichyn kehittämää lohkonsiirtoalgoritmia, jonka avulla kokonaisia lohkoja voidaan käsitellä kokonaisuuksina.

Vaikka RCS on tunnettu sovellus, sille ei ole olemassa virallista ohjekirjaa. Sen perusteet esitellään kuitenkin esimerkiksi Tichyn RCS-artikkelin [65] muokatussa versiossa[9] , joka toimitetaan RCS-jakelupaketin mukana troff-muodossa[10].

4.5.2 Toiminta

RCS käsittelee yksittäisiä tiedostoja, joiden eri versiot tallennetaan tiedoston nimestä johdettuun *,v*-päätteiseen tiedostoon. Esimerkiksi tiedoston `main.c` kaikki versiot tallennetaan `main.c,v`-nimiseen tiedostoon. Mikäli hakemistossa on *RCS*-niminen alihakemisto, versiotiedosto tallennetaan sinne.

Tiedosto laitetaan versionhallintaan komennolla `ci main.c`, joka luo uuden tiedoston `main.c,v` ja poistaa alkuperäisen. Tiedoston saa ulos versionhallinnasta komennolla `co main.c`, mutta oletuksena palautettu tiedosto on kirjoitussuojattu, joten sitä ei voi muuttaa. Tiedoston saa muokkauskelpoiseksi merkitsemällä se ulosoton yhteydessä lukituksi, mikä saadaan aikaan antamalla `co`:lle optio *-l*. Esitetty toiminta on aiemmin esitetyn lukitse-muokkaa-vapauta-menetelmän (s. 20) mukainen.

Tehtyjä muutoksia voidaan tutkia komennon `rcsdiff main.c` näyttämän diff-tulosteen avulla. Lopuksi muutokset voidaan tallentaa versionhallintaan samalla komennolla, millä alkuperäinen tiedosto laitettiin sinne. Versiohistorian tarkasteluun RCS tarjoaa komennon `rlog`.

[9] http://www.cs.purdue.edu/homes/trinkle/RCS/rcs.ms (viitattu 2.8.2007).
[10] http://www.troff.org/ (viitattu 14.9.2007).

4.5.3 Nykytila

Aikoinaan RCS oli merkittävä sovellus, ja, kuten Burns ym. asian ilmaisevat, "[RCS] is the definitive previous work in version control". [6]. Sen nykytilan selvittelyä varten käytiin läpi koko bug-rcs-listan[11] viestiarkisto sen perustamisesta 19.9.2000 aina nykyhetkeen (24.10.2007) asti. Suurin osa arkiston sisällöstä oli roskapostia, mutta joitakin asiallisiakin viestejä vielä löytyi. Niiden pohjalta voi kuitenkin selvästi huomata, että RCS on jo vanha ohjelma, joten siihen ei enää juurikaan tehdä muutoksia.

Odotetusti RCS-listalta ei löytynyt yhtään uutta innovaatiota, eikä edes yhtään virallista julkaisua ole tehty tämän vuosituhannen puolella. Ainoat merkittävät viestit käsittelivät RCS:n rajoitusta käyttäjätunnusten suhteen (ne eivät saa alkaa numerolla tai sisältää välilyöntejä). Koska tälle ja muille ongelmille ei kuitenkaan aiota enää tehdä mitään, voidaan sanoa, että RCS on viimeinkin tullut tiensä päähän.

4.6 CVS

CVS[12] (Concurrent Versions System) [13] [4, 23] on Dick Grunen ja hänen kahden oppilaansa alun perin RCS:n päälle kehittämä kokoelma komentotiedostoja, jotka mahdollistivat RCS-tietovaraston käytön monen kehittäjän projekteissa.

Alun perin CVS vaati RCS:n toimiakseen, mutta myöhemmin se irtautui siitä, säilyttäen silti yhteensopivuuden sen kanssa. Vuonna 1989 Brian Berliner kirjoitti CVS:n uudestaan C-kielellä ja myöhemmin Jeff Polk lisäsi siihen joitakin ominaisuuksia. [4, 17]

4.6.1 Ominaisuudet

CVS:n tarkoituksena on mahdollistaa usean kehittäjän samanaikainen toiminta. Tavoite saavutetaan mm. seuraavien ominaisuuksien avulla:

Koko hakemistorakenteen versiointi Muutokset, jotka vaikuttavat useaan tiedostoon, voidaan tallentaa samanaikaisesti.

Kehityshaarat Suuret muutokset voidaan tehdä erillisessä kehityshaarassa, josta muutos on myöhemmin helppo yhdistää perusversioon.

[11] http://lists.gnu.org/archive/html/bug-rcs/

[12] http://www.nongnu.org/cvs/ (viitattu 1.8.2007).

[13] Yleisesti käytetty muoto on myös Concurrent Versioning System.

Lukitusten välttäminen Tiedostoja ei tarvitse lukita muutosten ajaksi. Sen sijaan samaan tiedostoon tehdyt muutokset pitää yhdistää, ennen kuin ne voi tallentaa tietovarastoon.

Keskitetty tietovarasto Keskitetyn tietovaraston avulla pääsynvalvonta ja muutoksista ilmoittaminen voidaan hoitaa helposti.

CVS on GPL-lisenssin alainen vapaa ohjelma, joten sitä on käytetty paljon etenkin vapaiden ohjelmien kehityksessä. Suosionsa perusteella CVS:ää onkin jo pitkään pidetty versionhallintajärjestelmien de facto -standardina, mutta nykyisten uuden sukupolven versionhallintaohjelmien tulon myötä tämä asema on alkanut horjua.

4.6.2 Toiminta

CVS tallentaa tiedostot tietovarastoon RCS:n käyttämässä *,v*-muodossa, jossa jokaista tiedostoa vastaa oma tiedostonsa. Esimerkiksi tiedoston `oma.c` kaikki versiot tallennetaan tietovarastossa olevaan tiedostoon nimeltä `oma.c,v`. Tietovaraston rakenne vastaa työhakemiston rakennetta siten, että jokaista lähdekooditiedostoa vastaa oma *,v*-tiedosto. Työhakemistossa metadataa säilytetään jokaiseen hakemistoon luodussa *CVS*-nimisessä alihakemistossa.

Tiedostot laitetaan versionhallintaan komennolla `cvs add`. Tehtyjä muutoksia voidaan tarkastella komennon `cvs diff` avulla. Muutosten jälkeen uusi versio voidaan tallentaa tietovarastoon komennolla `cvs commit`. Tallennuskomennon yhteydessä voidaan antaa seloste muutoksen syystä.

4.6.3 Ongelmat

Vaikka CVS onkin suosittu, sen ikä alkaa jo näkyä. Suuresta käyttäjämäärästä johtuen ohjelmasta on löytynyt muutamia heikkouksia:

- Muutokset (commit) eivät olleet atomisia: jos muutos vaikutti useaan hakemistoon, ei voitu taata, ettei joku toinen muuttanut toisen hakemiston sisältöä samalla, kun toinen hakemisto oli lukittu.
- Hakemistoja ja tiedostoja ei voinut nimetä uudelleen. Kiertotiet olivat epäluotettavia.
- Pääsynvalvonta käytti tiedostojärjestelmän oikeuksia (vrt. Windows vs. Linux).

- Hakemistoja ei voinut versioida, eikä tyhjiä hakemistoja voinut olla.
- Kehityshaarojen käyttö oli hankalaa.
- Työskentely ilman verkkoyhteyttä oli rajoitettua.

Nämä ongelmat alkoivat näkyä siinä vaiheessa, kun projektit kasvoivat riittävän isoksi. CVS oli kuitenkin niin yleinen, että sitä käytettiin ongelmista huolimatta.

4.6.4 Merkitys

1990-luvun alkupuolella tapahtui asioita, jotka vaikuttivat merkittävästi versionhallinnan käyttöön vapaiden ohjelmien kehittämisessä. RCS siirtyi GPL-lisenssin alaisuuteen ja CVS alkoi tukea verkon käyttöä: Jim Kingdonin vuonna 1990 lisäämät verkko-ominaisuudet mahdollistivat kehittäjille pääsyn versionhallintaohjelman tietovarastoon mistä tahansa Internetin avulla; tämä ja monet muut muutokset johtivat lopulta RCS:stä itsenäistyneen CVS-II:n syntyyn. [17]

Näihin aikoihin myös vapaiden ohjelmien käsite nousi yleiseen tietoisuuteen mm. Linuxin saavuttaman suosion ansiosta. Tietoverkkojen käyttäminen näiden ohjelmien kehittämiseen oli luonnollinen valinta, sillä kehittäjät olivat hajaantuneet maailmanlaajuisesti. Suurin osa kehitysprojekteista alkoikin käyttää uudistunutta CVS:ää sen lukuisista puutteista huolimatta, mikä vakiinnutti sille lopulta de facto -standardin aseman. [8]

Internetin yleistyminen vuosikymmenen loppupuolella kasvatti CVS:n suosiota entisestään, antaen sille lentävän lähdön seuraavalle vuosituhannelle. Moni palveluntarjoaja alkoi tarjota tallennustilaa ja kehitystyökaluja CVS-projekteille[14]. Tämä oli hyödyllistä siitäkin syystä, että CVS vaati keskitetyn tietovaraston.

4.6.5 Nykytila

CVS:n tarkoituksena oli tarjota versionhallintamahdollisuudet monen käyttäjän projekteille. Yleisesti käytetty RCS ei siihen sellaisenaan soveltunut, etenkin kun se käytti aiemmin esiteltyä *lukitse-muokkaa-vapauta*-tyylistä lukitusjärjestelmää (s. 20). CVS:n omintakeinen ratkaisu tähän ongelmaan oli *kopioi-muokkaa-yhdistä*-menetelmä (s. 21), joka mahdollisti usean kehittäjän toiminnan samanaikaisesti. Nykyisin tämä menetelmä on käytössä lähes kaikissa versionhallintaohjelmissa.

[14] http://www.ibiblio.org/fosphost/exhost.htm (viitattu 13.11.2006).

Vaikka CVS on jo vanha järjestelmä, de facto -standardin asemaa pitkään kantaneena sillä on vielä monia käyttäjiä. Luotettavaksi osoittautuneena se on aivan käyttökelpoinen ohjelma vielä nykyäänkin, sillä sen ominaisuudet ja ongelmat tunnetaan hyvin. Yleisyytensä vuoksi CVS on tunnettu ja hyvin dokumentoitu, joten monille vasta-alkajille se on edelleenkin helpoin tie versionhallinnan maailmaan.

CVS:n asema versionhallintaohjelmien käytännön standardina on jo vakavasti horjunut, ja tällä hetkellä sen manttelinperijäksi kaavailtu Subversion näyttäisikin saavuttaneen tavoittelemansa aseman. Se ei ole kuitenkaan yksin, sillä CVS:n seuraajaksi löytyy myös muita varteenotettavia vaihtoehtoja. Mitään kiirettä vanhojen järjestelmien korvaamiseen ei kuitenkaan vielä ole, sillä vaikka CVS:n kehitys ei enää tuokaan mitään uutta, sillä on vielä vankka kannattajakuntansa, ja jo olemassa olevien toimintojen laaja tuki muissa ohjelmissa lupaa tälle ohjelmalle vielä joitakin elinvuosia.

4.7 Aegis

Aegis [46][15] on Peter Millerin kehittämä transaktiopohjainen konfiguraationhallintajärjestelmä. Perinteisistä konfiguraationhallintaohjelmista poiketen se ei sisällä lainkaan omia versionhallintaominaisuuksia, vaan jättää niihin liittyvät toiminnot muiden sovellusten, kuten RCS:n, vastuulle. Aegiksen tehtävä onkin lähinnä täydentää olemassa olevien versionhallintajärjestelmien toiminnallisuutta niin, että ne soveltuvat myös konfiguraationhallintaan.

Miller kertoo Aegiksen syntyvaiheista seuraavaa[16]:

> The idea for Aegis did not come full-blown into my head [...] but rather from working in a software shop which used a simplistic form of something similar. [...] It turns out that the system used is nothing new, and is described in many SCM textbooks; it is the result of systematically resolving development issues for large-ish teams.

Teoreettiselta kannalta tarkasteltuna Aegis ei tuonutkaan mukanaan mitään uutta, mutta 1990-luvun alussa konfiguraationhallintajärjestelmä oli harvinainen tapaus avoimen lähdekoodin versionhallintaohjelmien joukossa. Vuosien saatossa sitä on myös ajanmukaistettu niin, että vuonna 1997 siihen tuli tuki erillisille kehityshaa-

[15] http://aegis.sourceforge.net/ (viitattu 26.7.2007).

[16] Aegis 4.22:n lähdekoodipaketissa oleva README-tiedosto.

roille ja 1999 tuki hajautetulle kehitykselle. Vuonna 2004 Aegikseen tuli mahdollisuus tunnistaa tiedostoja UUID:n (Universally Unique IDentifier) avulla, mikä mahdollisti mm. tiedostojen uudelleennimeämisten paremman käsittelyn.

4.7.1 Ominaisuudet

Aegis on Unix-periaatteiden mukaisesti toimiva järjestelmä, joka on suunniteltu toimimaan muiden ohjelmien kanssa[17]. Ohjelma toteuttaa muutostenhallintatoiminnallisuuden käyttämällä järjestelmän omia sovelluksia toimintansa apuna.

Aegis hallinnoi tietovarastossa olevaa lähdekoodia, jota kutsutaan *perusversioksi* (engl. baseline). Perusversion on oltava aina toimintakunnossa, joten ohjelman tehtävänä on rajoittaa siihen kohdistuvia muutoksia. Rajoitus tapahtuu seuraavien ominaisuuksien avulla:

- Vain Aegiksella itsellään on kirjoitusoikeudet tietovarastoon.
- Jokaisen muutoksen mukana pitää olla testi, joka epäonnistuu ilman muutosta ja onnistuu muutoksen kanssa.
- Jokainen muutos läpikäy kuusi askelta, ennen kuin se hyväksytään.
- Kehittäjät kuuluvat eri rooleihin, joten muutoksen tehnyt henkilö ei voi itse arvioida omaa tuotostaan.

Vaikka järjestelmä onkin hyvin muodollinen, Aegis on pyritty suunnittelemaan sellaiseksi, ettei se ole koko ajan tiellä. Käyttäjät on huomioitu mm. ohjelman antaman riittävän palautteen avulla.

Aegis on lisensoitu GPL:n alaisuuteen.

4.7.2 Toiminta

Koska Aegis on varsinaisesti konfiguraationhallintaohjelma, se keskittyy versionhallinnan sijaan laajaan ohjelmistoon tehtävien muutosten hallinnointiin. Aegiksen toimintafilosofia perustuu ajatukseen koko ajan vakaana pysyvästä perusversiosta, joka päivitetään aina atomisesti uudeksi vakaaksi versioksi. Päivitysten vakaus varmistetaan siten, että jokaista niihin liittyvää muutosta varten vaaditaan testitapaus, jonka pitää epäonnistua aikaisemmassa ja onnistua uudemmassa versiossa.

[17] http://mysite.verizon.net/ralph.a.smith1/aegis/user-guide-html/ug-introduction.html (viitattu 6.11.2007).

Kaikki Aegiksen toiminnot ovat change set -pohjaisia, eli eri tiedostoihin tehtävät muutokset kootaan yhdeksi muutosjoukoksi. Kehityksensä aikana tällaiset muutosjoukot voivat olla jossakin seuraavista kuudesta tilasta [46][18]:

Odottaa kehitystä (engl. Awaiting Development): Muutos aloittaa elinkaarensa tässä tilassa, kun se on hyväksytty toteutettavaksi. Seuraavaan tilaan ei voida siirtyä, ennen kuin joku kehittäjä ottaa vastuun suunniteltujen muutosten toteuttamisesta. Tämä tila on kaikille muutoksille pakollinen, eli tätä ei voi missään tapauksessa ohittaa. Edes ylläpitäjä ei voi määrätä toteutusvastuuta suoraan jollekin kehittäjälle, sillä se heikentäisi heidän valinnanmahdollisuuksiaan.

Kehityksessä (engl. Being Developed): Muutos on tässä tilassa silloin, kun sille on saatu vastuullinen kehittäjä. Kaikki muutokseen liittyvä kehitys tapahtuu tässä tilassa, mutta niin, että vain kehittäjillä on oikeus muokata tiedostoja. Tarkastajat ja integraattorit eivät voi muuttaa mitään, mutta heillä on niin päättäessään oikeus keskeyttää kehitys ja hylätä sen tulokset. Seuraavaan tilaan päästäkseen muutoksen pitää kääntyä, sisältää testejä ja päästä nämä testit läpi.

Odottaa katselmointia (engl. Awaiting Review): Muutos siirtyy tähän tilaan silloin, kun kehittäjät katsovat sen olevan valmis. Tavallisesti tätä tilaa ei käytetä pienissä projekteissa, mutta isompien kohdalla tätä voidaan käyttää katselmointien koordinointiin. Aegis tarjoaa mm. mahdollisuuden lähettää sähköpostia katselmoijille aina silloin, kun jokin muutos siirtyy tähän tilaan.

Katselmoitavana (engl. Being Reviewed): Muutos on tässä tilassa, kun sen kehitys on päättynyt ja se on otettu katselmoitavaksi. Muutoksesta tarkistetaan, vastaako sen toiminnallisuus annettua kuvausta, ja onko lähdekoodi käytetyn standardin mukaista. Aegis estää kehittäjää tarkistamasta omaa koodiaan, mikä takaa sen, että ainakin yksi ulkopuolinen henkilö on tutustunut muutokseen. Tämä esto voidaan kuitenkin ottaa pois päältä, mutta niin suositellaan tehtäväksi vain yhden (tai kahden) hengen projekteissa.

[18] Millerin lähteessä [46] esittämien tilojen joukosta puuttuu kohta *Odottaa katselmointia* (Awaiting Review), joka on otettu käyttöön myöhemmin. http://mysite.verizon.net/ralph.a.smith1/aegis/user-guide-html/ug-c7_0-how_aegis_works.html (viitattu 4.10.2007).

Odottaa integrointia (engl. Awaiting Integration): Muutos päätyy tähän tilaan silloin, kun tarkastajat ovat hyväksyneet sen. Tässä tilassa katselmoidut muutokset jonottavat pääsyään integrointiin, sillä perusversioon niitä voidaan yhdistää vain yksi kerrallaan.

Integroitavana (engl. Being Integrated): Muutos on tässä tilassa silloin, kun se on otettu integroitavaksi perusversioon. Aluksi perusversiosta tehdään kopio, johon käsiteltävä muutos yhdistetään. Tämän jälkeen lopputulos käännetään ja testataan. Integroija voi hylätä muutoksen joko testien epäonnistumisen tai jonkin muun syyn vuoksi, joten tämä tila voi toimia myös toisena arviointitilana. Myös katselmointien laatua voidaan arvioida tässä tilassa.

Jos muutos selviää onnistuneesti kaikkien näiden vaiheiden läpi, se pääsee tilaan **Valmis** (engl. Completed). Tässä tilassa muutos on tullut osaksi vakaata perusversiota, joten sitä ei voida enää perua.

4.7.3 Nykytila

Aegiksen aiempien vaiheiden ja nykytilan selvittämistä varten käytiin läpi aegis-developers-sähköpostiarkiston[19] viestit ajanjaksolta 1.1.2005–30.9.2007.

Tammikuussa 2005 Peter Miller tiedusteli, onko kenelläkään kokemuksia avoimen lähdekoodin XML-parserointikirjastoista. Hän tarvitsi sellaista, koska oli tekemässä tukea muutosjoukkojen (engl. Change Set) lähettämiseen ja vastaanottamiseen RevML:n (Revision Markup Language)[20] kautta. Vastaajat mainitsivat expatin ja libxml2:n, jonka jälkeen keskustelu hiipui. Seuraavat viestit ovat huhtikuulta 2005 ja käsittelevät kääntöongelmaa.

Vuonna 2006 listalle tuli vain yksi viesti, jossa kerrottiin Aegiksen www-sivuille tehdyistä parannuksista. Vuoden 2007 alussa viestiliikenne taas kasvoi, ja keskustelua käytiin mm. Boost-kirjaston[21] käytöstä ja uuteen GPLv3-lisenssiin siirtymisestä. Lisenssikeskustelun ilmapiiri oli harvinaisen rauhallinen, eikä kukaan vastustanut uuteen lisenssiversioon siirtymistä. Jonkin verran keskustelua aiheutti kysymys siitä, kuinka paljon GPL:n käyttö BSD-lisenssin sijasta vaikutti Aegiksen menestykseen.

Viestien määrä pysyi matalana koko alkuvuoden ajan, mutta syksyllä kehittäjien

[19] http://sourceforge.net/mailarchive/forum.php?forum_name=aegis-developers
[20] http://public.perforce.com/public/revml/index.html (viitattu 9.10.2007).
[21] http://www.boost.org/ (viitattu 9.10.2007).

aktiivisuus taas lisääntyi. Aegiksen versio 4.22.2 julkaistiin 18.10.2007, ja sen tulevaisuudesta käytiin jonkin verran keskustelua. Mikäli suunnitelmat toteutuvat, Aegiksesta tulee GITin jälkeen toinen järjestelmä, joka toteuttaa myöhemmin esiteltävän innovatiivisen bisect-toiminnallisuuden.

4.8 Vesta

Vesta [27, 28][22] on Digital Equipment Corporation (DEC) -yrityksen sisäiseen käyttöönsä kehittämä konfiguraationhallintajärjestelmä. Sen kehitys alkoi 1990-luvun alussa, mutta vuosikymmenen puolivälissä se kirjoitettiin kertyneen kokemuksen perusteella kokonaan uusiksi.

Vuonna 1998 Vesta otettiin käyttöön DECin Alpha-prosessorien suunnittelussa, ja sen oikeudet siirtyivät DECin kanssa kauppoja tehneelle Compaqille. Kun Compaq vuonna 2001 myi Alpha-prosessorien oikeudet Intelille, Vesta julkaistiin avoimen lähdekoodin LGPL-lisenssin alaisuudessa.

4.8.1 Ominaisuudet

Vesta on suunniteltu toimivaksi kaikenkokoisten kehitysprojektien kanssa. Ohjelma on osoittautunut vakaaksi ja tehokkaaksi, ja sillä on takanaan yli kymmenen vuoden mittainen historia. Kehityksen tuloksena Vesta tarjoaa seuraavat ominaisuudet[23]:

Automaattinen riippuvuuksien havaitseminen (engl. automatic dependency detection): Riippuvuuksien havaitseminen on ohjelmointikielestä riippumatonta, tarkkaa ja yksityiskohtaista. Jopa sellaiset yksityiskohdat, kuten kääntäjän versionumero, käännös- ja linkitysoptiot sekä käytettyjen järjestelmäkirjastojen tiedot tallennetaan. Käyttäjän ei tarvitse huolehtia riippuvuuksien oikeellisuudesta, sillä ne hoidetaan automaattisesti. Aikamerkinnät eivät vaikuta riippuvuuksiin, joten järjestelmän aikatietojen muuttuminen tai virheelliset tulostiedostot eivät aiheuta ongelmia.

Käyttäjäriippumattomat kokoamiset (engl. user-independent builds): Ohjelmien kokoamiset tapahtuvat eristetyissä ympäristöissä, joihin käyttäjillä ei ole pääsyä. Tämän vuoksi prosessin kulkuun ei voi vaikuttaa ulkopuolelta, joten se on täysin ennakoitavissa ja toistettavissa.

[22] http://www.vestasys.org/ (viitattu 1.8.2007).

[23] http://www.vestasys.org/why-vesta.html (viitattu 9.10.2007).

Taattu kokoamisten toistettavuus (engl. guaranteed repeatability of builds): Ohjelman koonnissa kaikki lopputulokseen vaikuttavat seikat on tarkasti määritelty. Näitä seikkoja ovat mm. lähdekooditiedostojen, kääntäjien ja käytettyjen kirjastojen versiot. Näiden avulla jokainen aiemmin tehty kokoaminen on mahdollista toistaa täydellisesti siten, että lopputulos on identtinen. Tästä on apua erityisesti virheiden korjaamisessa ja laadunvarmistuksessa.

Yhteistyöominaisuudet (engl. collaboration features): Lähdekoodin eri versioita ja jopa kokonaisia kokoonpanoja (engl. builds) voidaan helposti jakaa Vestan replikaattorin avulla. Taattu kokoamisen toistettavuus on voimassa myös jaettujen kopioiden kanssa, vaikka kokoaminen tapahtuisikin eri järjestelmässä. Tietovaraston käyttöliittymätyökalut on suunniteltu toimimaan myös muualla kuin paikallisesti sijaitsevien tietovarastojen kanssa. Tietojen ulosotto paikalliseen työhakemistoon onnistuu etätietovarastosta, mikäli sinne on sopivat oikeudet. Muutosten tallentaminen puolestaan kopioi uuden version oikeaan paikkaan.

Tiedostojärjestelmärajapinta tietovarastoon (engl. file-system interface to the repository): Kaikkiin versioihin pääsee käsiksi tiedostojärjestelmän kautta, joten käyttäjä voi käsitellä tiedostoja järjestelmän perustyökalujen avulla. Tiedostoista voi esimerkiksi etsiä tekstiä *grep*-ohjelmalla ja niitä voi vertailla *diff*-ohjelmalla. Osaavat käyttäjät voivat tehdä myös omia skriptejä, mikäli näkevät niille tarvetta.

Jaettu kokoamistietojen välimuisti (engl. shared cache of build results): Koska kokoaminen on käyttäjäriippumatonta, eri käyttäjien saamat tulokset voidaan jakaa muiden käyttäjien kesken, jolloin turha päällekkäinen työ jää pois. Kaikkien samassa paikassa (engl. site) olevien kehittäjien työn tulokset tallennetaan yhteiseen välimuistiin, joten ne ovat suoraan kaikkien hyödynnettävissä.

Se on tehokas (engl. it's high-performance): Vestan kokoamistoiminto on nopeampi kuin perinteinen *make*-ohjelma. Se on monisäikeinen ja voi suorittaa useita ohjelmia samanaikaisesti jopa eri tietokoneissa. Tietojen ulosotto tietovarastosta on vakioaikainen toiminto, koska tietovarastossa on jokaisen lähdekooditiedoston kokonainen kopio. Tiedostojen kopiointiin ei kulu aikaa, koska tietovaraston tiedostojärjestelmä hoitaa työkopioiden hallinnan. Ulosotettu tiedosto

ei kuluta levytilaa ennen kuin sitä on muutettu, sillä tietovarasto tukee *copy-on-write*-toimintoa.

Se on vapaa (engl. it's free): Vesta on vapaa ohjelma, jonka lisenssinä on GNU Lesser General Public License .

Se on laajennettava (engl. it's extensible): Komentorivityökalujen avulla on helppo kirjoittaa skriptejä monimutkaisten tehtävien avuksi. Tietovaraston sisältöön voidaan liittää kaikenlaista metatietoa joustavan attribuuttijärjestelmän ansiosta. Mikäli valmiit työkalut eivät riitä, uusia komentoja voidaan kirjoittaa C++-kielellä käyttämällä Vestan ohjelmointirajapintaa. Koska käytettävä lisenssi on LGPL, laajennosten tekijä voi päättää itse oman koodinsa lisenssiehdoista.

Se on kypsä (engl. it's mature): Vestaa on tutkittu ja kehitetty 1990-luvun alkupuolelta lähtien ja se on ollut vaativassa tuotantokäytössä useiden vuosien ajan. Se on hyvin testattu, vakaa, skaalautuva ja vikasietoinen. Vestan suunnittelussa on alusta alkaen otettu huomioon olemassa olevien järjestelmien puutteet, joten se on niihin verrattuna hyvin kilpailukykyinen myös kaupallisessa mielessä.

Vesta on varsinaisesti konfiguraationhallintajärjestelmä, mutta, toisin kuin Aegis, se hoitaa itse myös oman versionhallintansa. Sen suurin puute on siitä puuttuva oma yhdistämistuki, joka pitää hoitaa ulkopuolisella skriptillä. Tähän tarkoitukseen suositellaan Precise Codeville Merge -algoritmia käyttävää *vmerge2.py*:tä[24].

4.8.2 Toiminta

Vesta vaatii melko paljon konfigurointia, ennen kuin sen voi ottaa käyttöön. Alkuvalmisteluiden jälkeen järjestelmä on kuitenkin helppokäyttöinen. Kaikki toiminta tapahtuu hakemistojen */vesta* ja */vesta-work* alla.

Uusi projekti luodaan komennolla `vcreate`, joka tekee uuden hakemiston */vesta*-hakemiston alle.

Projektin muokkaaminen aloitetaan ottamalla se ulos erilliseen työhakemistoon komennolla `vcheckout`, joka kysyy tietoja muokkauksen syistä ja luo työhakemiston */vesta-work*-hakemiston alle. Kaikki muokkaukset tehdään työhakemistossa.

[24]http://wiki.vestasys.org/MergingFuture/DevPlan (viitattu 4.11.2007).

Tiedostojen muokkaus tapahtuu perinteisillä tavoilla. Tehdyistä muutoksista voidaan ottaa tilannekuvia (engl. snapshot) komennon `vadvance` avulla. Komento tallentaa tilannekuvan */vesta*-hakemiston alle. Muutoksia voidaan tarkastella vertaamalla alkuperäistä versiota tilannekuvaan.

Kun kaikki muutokset on tehty, komento `vcheckin` pyytää muutoksen selosteen ja tekee viimeisimmästä tilannekuvasta uuden version. Samalla työhakemisto häviää. Seuraavat muutokset aloitetaan uudella `vcheckout`-komennolla.

Komennon `vbranch` avulla voidaan käyttää kehityshaaroja. Kehityshaarojen yhdistämiseen ei toistaiseksi ole omia työkaluja, mutta tähän voidaan käyttää Unix-järjestelmien perinteisiä `diff`- ja `patch`-ohjelmia ja aiemmin mainittua `vmerge2.py`-skriptiä.

4.8.3 Nykytila

Nykytilan selvittämistä varten käytiin läpi Vestan kehityslistan sähköpostiarkiston[25] viestit ajanjaksolta 1.1.2005–30.9.2007.

Vestan uusin julkaistu versio on 2.1.12.pre13-10, joka julkaistiin elokuussa 2007. Julkaisussa on bugikorjausten lisäksi keskitytty ohjelman siirrettävyyteen[26]. Myös seuraava versio on jo suunnitteilla, joten ohjelman kehitys näyttää turvatulta.

Vesta on vanha ja vakaa järjestelmä, joka on osoittanut toimivuutensa. Vaikka se ei tarjoakaan tukea uusimpien versionhallintaohjelmien tarjoamille toiminnoille, se on täysin käyttökelpoinen konfiguraationhallinnan tarpeisiin.

4.9 PRCS

PRCS (Project Revision Control System) [42][27] on Paul N. Hilfingerin avustajiensa Luigi Semenzaton ja Joshua MacDonaldin kanssa suunnittelema eri versionhallintaohjelmien päällä toimiva järjestelmä, joka mahdollistaa useiden tiedostojen ja hakemistojen käsittelyn kokonaisuuksina. Sovelluksen varsinaisesta toteutuksesta vastasi MacDonald, joka on nykyisin myös sen ylläpitäjä.

[25] https://sourceforge.net/mailarchive/forum.php?forum_name=vesta-devel

[26] https://sourceforge.net/project/shownotes.php?group_id=34164&release_id=529707 (viitattu 1.11.2007).

[27] http://prcs.sourceforge.net/ (viitattu 15.8.2007).

4.9.1 Ominaisuudet

PRCS:n tavoitteet ovat yhtenevät muiden vastaavanlaisten versionhallintaohjelmien kanssa, mutta tekijöidensä mielestä se on muita järjestelmiä huomattavasti yksinkertaisempi.

Ensimmäinen PRCS:n versio käytti RCS:ää tietovarastonaan. PRCS:n toisen version kehitys aloitettiin vuonna 1997, tavoitteena toteuttaa tuki asiakas-palvelin-mallille. Tämän projektin ensimmäisenä tuloksena oli *Xdelta*, joka on *rsync*-algoritmiin perustuva deltojenpakkausmenetelmä.

PRCS:n käyttämä käsite *projekti* tarkoittaa kokonaisia hakemistoja ja niiden sisältämiä tiedostoja kaikkine versioineen. Jokaiseen versioon kuuluu ns. *versiokuvaustiedosto* (engl. version description), joka määrittää kyseiseen versioon kuuluvat tiedostot. Versiokuvaustiedosto nimetään projektin mukaan, eli Proj-nimisen projektitiedoston versiokuvaustiedosto on nimeltään *Proj.prj*.

PRCS säilyttää projekteja erillisessä tietovarastossa. Muokkausta varten projekti pitää ottaa ulos erilliseen työhakemistoon.

PRCS on GPL-lisenssin alainen vapaa ohjelma. Sen käyttöjärjestelmätuki on rajallinen, eikä tukea esimerkiksi Windows- tai Macintosh-järjestelmille ole edes näköpiirissä.

4.9.2 Toiminta

Uusi projekti aloitetaan komennolla `prcs checkout Proj`. Komento tekee oletushakemistosta työhakemiston ja lisää sinne *Proj.prj*-nimisen versiokuvaustiedoston. Tiedoston rakenne on Lisp-kielen syntaksin mukainen.

Projektiin lisätään uusia tiedostoja lisäämällä niiden kuvaukset versiokuvaustiedostoon. Tämän voi tehdä käsin, mutta helpompi tapa on käyttää PRCS:n tarjoamaa komentoa `prcs populate`, joka mahdollistaa yksittäisten tiedostojen tai vaikka koko hakemistorakenteen lisäämisen samanaikaisesti.

Lisäysten jälkeen versiokuvaustiedostoon voidaan kirjoittaa kommentti tehdyistä lisäyksistä. Lopuksi uusi versio tallennetaan komennolla `prcs checkin`. Muutosten tallentaminen tapahtuu samalla tavalla, kuin uusien tiedostojen lisäys. Tehtyjä muutoksia voidaan tarkastella komennolla `prcs diff`.

Mikäli tietovarastoon on tullut uusia versioita, muutosten tallentaminen ei onnistu ennen yhdistämistä. Yhdistäminen tapahtuu komennolla `prcs merge`, jonka jälkeen työhakemistossa oleva yhdistetty versio voidaan tallentaa, kuten mikä ta-

hansa muutos.

4.9.3 Nykytila

Nykytilan selvittelyä varten käytiin läpi projektin ainoan virallisen sähköpostiarkiston[28] kaikki viestit ajalta 5.2.2005–21.7.2007. Viimeisin julkaistu versio on 1.3.3 vuoden 2004 toukokuulta. Julkaisun jälkeen ohjelman edistymisestä ei ole mitään tietoja. Keskustelufoorumille on kertynyt seitsemän vuoden aikana alle kymmenen viestiä, joten aktiivisia käyttäjiä ei näytä olevan. Viimeisissä viesteissä kysellään, onko projekti kuollut[29]. Vastaus näyttää myönteiseltä.

4.10 Subversion

Subversion on Ben Collins-Sussmanin ja Karl Fogelin kehittämä keskitetty versionhallintaohjelma. Se sai alkunsa vuonna 2000, kun CollabNet, Inc[30] aloitti projektin, jonka tarkoituksena oli suunnitella ja toteuttaa vanhalle mutta paljon käytetylle CVS:lle seuraaja, joka korjaisi sen pahimmat puutteet, mutta olisi muuten sen kaltainen.

Helmikuussa 2000 CollabNet otti yhteyttä Fogeliin, tunnettuun CVS-asiantuntijaan, joka oli kirjoittanut CVS:n käyttöä avoimen lähdekoodin projekteissa käsittelevän kirjan [17]. Hän oli jo aiemmin miettinyt ystävänsä Jim Blandyn kanssa uuden versionhallintaohjelman tekemistä, joten he suostuivat tehtävään. [8] Blandy sai työnantajaltaan Red Hatilta luvan käyttää työaikansa uuden järjestelmän kehittämiseen, ja Fogel siirtyi CollabNetin palvelukseen. CollabNet pestasi mukaan myös Collins-Sussmanin, ja yhdessä nämä kolme laativat Subversionin ensimmäiset suunnitelmat[31].

Subversionin toteutus alkoi kesäkuussa 2000, ja elokuussa 2001 se oli siinä kunnossa, että sen kehityksessä voitiin siirtyä sen itsensä käyttämiseen[32]. Monissa lähteissä Subversionin julkaisuvuotena pidetään vuotta 2001, mutta ainakin tämän tutkielman kannalta vuosi 2000 on oikeampi, sillä sen kehitystä on voinut seurata siitä

[28] http://sourceforge.net/mailarchive/forum.php?forum_name=prcs-cvs-list

[29] http://sourceforge.net/forum/forum.php?forum_id=36280 (viitattu 4.9.2007).

[30] http://www.collab.net/ (viitattu 11.10.2006).

[31] Collins-Sussmanin haastattelu, http://www.advogato.org/article/786.html (viitattu 22.10.2007.).

[32] http://subversion.open.collab.net/articles/SubversionHistory.html (viitattu 27.10.2007).

lähtien.

4.10.1 Ominaisuudet

Subversion on alusta alkaen suunniteltu CVS:n seuraajaksi, joten sen kehittelyssä on otettu huomioon kaksi pääkohtaa: sen on oltava mahdollisimman paljon CVS:n kaltainen ("look and feel"), mutta samalla korjattava CVS:n pahimmat puutteet [8][33]. Tavoitteena oli, että siirtyminen CVS:stä Subversioniin olisi mahdollisimman vaivatonta.

Subversion tarjoaa seuraavat parannukset CVS:ään nähden:

Hakemiston versiointi CVS hallinnoi vain tiedostojen historiaa. Subversion hallinnoi tiedostojen lisäksi myös kokonaisten hakemistojen historiaa.

Todellinen versiohistoria CVS ei säilyttänyt tietoja tiedostojen kopioinnista ja uudelleennimeämisistä. Subversion tukee sekä tiedostojen ja hakemistojen poistoja, lisäyksiä ja uudelleennimeämisiä.

Atomiset tallennukset Tehdyt muutokset tallentuvat tietovarastoon joko kaikki kerralla tai ei ollenkaan. CVS:ssä saattoi käydä niin, että vain osa muutoksista tallentui.

Metadatan versiointi Tiedostoihin ja hakemistoihin voidaan liittää ns. ominaisuuksia, eli avain-arvo-pareja. Ominaisuudet versioidaan samalla tavalla kuin muutkin tiedot.

Tehokkaat kehityshaarat Subversionin kehityshaarat ovat tavallisia perushakemiston kopioita, jotka tehdään kovia linkkejä tai vastaavia mekanismeja käyttämällä.

Subversion käyttää Apache/BSD-lisenssin kaltaista lisenssiä[34].

4.10.2 Toiminta

Subversionin tietovarastona toimii virtuaalinen puumainen tiedostojärjestelmä [8]. Samassa tietovarastossa voidaan säilyttää useiden projektien versiotietoja tallentamalla jokainen projekti omaan alihakemistoon.

[33] http://subversion.tigris.org/ (viitattu 13.8.2007).

[34] http://subversion.tigris.org/license-1.html (viitattu 24.9.2007).

Seuraavassa käyttöesimerkissä käytetään esimerkkinä Proj-nimistä kehitysprojektia. Tietovarasto puolestaan sijaitsee hakemistossa `/repo/`.

Olemassaoleva projekti laitetaan versionhallinnan alaisuuteen komennolla `svn import proj file:///repo/proj`. Yleisenä suosituksena on, että `proj`-hakemisto on valmisteltu etukäteen oikeanlaiseksi, eli sellaiseksi, että se sisältää alihakemistot `branches`, `tags` ja `trunk`. Näistä kaksi ensimmäistä on alussa tyhjiä ja viimeinen sisältää varsinaisen lähdekoodin.

Projektin muokkaus tapahtuu ottamalla ns. työkopio ulos tietovarastosta. Tämä tapahtuu komennolla `svn checkout file:///repo/proj work`. Komento luo `work`-nimisen hakemiston ja täyttää sen lähdekooditiedostoilla.

Tiedostoja voidaan muokata työhakemistossa. Tehtyjä muutoksia tarkastellaan komennolla `svn diff`. Lopuksi muutokset tallennetaan komennolla `svn commit`.

Perustoiminnot muistuttavat hyvin paljon CVS:n vastaavia toimintoja, mikä ei ole sattumaa. Tavoitteena oli nimenomaan taata CVS:n käyttäjien tyytyväisyys, mikä näyttääkin onnistuneen hyvin.

4.10.3 Hyppydeltat

Kuten aiemmin esitettiin (2.4.3, s. 14), perinteisten deltamenetelmien aikavaativuus on lineaarinen, mikä aiheuttaa ongelmia etenkin suurten projektien yhteydessä. Subversionissa näitä skaalautuvuusongelmia pyritään välttämään ns. *hyppydeltojen* (engl. skip delta) avulla, jolloin pitkien deltaketjujen läpikäynti voidaan välttää. [49]

Hyppydeltojen toiminta vastaa linkitetyn listan läpikäyntiä nopeuttavien *hyppylistojen* (engl. skip list) käyttöä, mutta on paljon helpompi toteuttaa: perusversioksi valitaan edellisen version sijaan yksinkertaisesti se versio, mikä saadaan nollaamalla käsiteltävänä olevan version versionumeron oikeanpuoleisin ykkösbitti.

Kuvasta 4.1 selvästi nähdään, että hyppydeltoja käyttämällä operaatioiden vaativuus laskee logaritmiseksi, mutta samalla tallennustilan tarve kasvaa. Tätä ei kuitenkaan ole pidetty ongelmana enää pitkään aikaan, koska – Brian Berlinerin sanoin – "[...] disk space is simply burned since it is 'cheap'." [4]. Nykyisin versionhallintaohjelmissa keskitytäänkin tilansäästön sijaan merkittäviksi katsottujen operaatioiden tehokkuuteen. [43]

4.10.4 Nykytila

Subversion suunniteltiin alusta alkaen CVS:n korvaajaksi, ja siinä se on onnistunut-

versio	perusversio	
0 = 0000	0000 = 0	
1 = 0001	0000 = 0	
2 = 0010	0000 = 0	
3 = 0011	0010 = 2	0 <- 1 2 <- 3 4 <- 5 6 <- 7
4 = 0100	0000 = 0	0 <------ 2 4 <------ 6
5 = 0101	0100 = 4	0 <---------- 4
6 = 0110	0100 = 4	0 <------------------------------ 8
7 = 0111	0110 = 6	
8 = 1000	0000 = 0	

Kuva 4.1: Perusversion valinta hyppydeltoissa.

kin varsin hyvin. Jos CVS oli ennen epävirallinen standardi, niin samaa voidaan sanoa nyt myös Subversionista. Sen nykyinen asema on kuitenkin koko ajan uhattuna, sillä sillä on haastajana monia uuden sukupolven hajautettuja versionhallintaohjelmia. Etenkin GIT ja Mercurial ovat kasvattaneet käyttäjäkuntaansa nopeasti, joten nähtäväksi jää, saako Subversion haalittua uusia käyttäjiä muualta kuin entisten CVS-käyttäjien joukosta. Haastajista huolimatta Subversionin asemaa voidaan kuitenkin toistaiseksi pitää turvattuna.

4.11 SourceJammer

SourceJammer[35] on Robert MacGroganin kehittämä asiakas-palvelin-arkkitehtuuriin perustuva lähdekoodinhallinta- ja versiointijärjestelmä. Se on kaksoislisensoitu sekä GPL:n että LGPL:n alle. Lisenssiratkaisua on perusteltu ohjelman FAQ:ssa (Frequently Asked Questions) seuraavasti[36]:

> The LGPL release of SourceJammer is provided for users who either have a philosophical problem with GPL or who want to include SourceJammer or some of SourceJammer's code into a proprietary system.
>
> The code in both the LGPL and GPL releases of SourceJammer is exactly the same except for one difference. The GPL release includes a slightly more robust text diff engine.

[35] http://www.sourcejammer.org/ (viitattu 1.8.2007).

[36] http://www.sourcejammer.org/psjs_faqs/08286780.html (viitattu 2.8.2007).

SourceJammer on kirjoitettu Java-kielellä, mikä näkyy mm. siinä, että se on toinen kahdesta tarkasteluun päätyneestä täysin graafisesta sovelluksesta, samalla kielellä toteutetun Superversionin ollessa toinen. Tekijänsä mukaan SourceJammerin ilmaisuus, yksinkertaisuus ja helppo asetusten säätö tekevät siitä varteenotettavan vaihtoehdon etenkin pienten ja keskisuurten ohjelmistokehitysprojektien versionhallintaohjelmaksi.

4.11.1 Ominaisuudet

Tyypillisen Java-ohjelman mukaisesti SourceJammerin ominaisuuksien esittelyssä on keskitytty enemmän näkyviin ominaisuuksiin kuin varsinaisiin ydintoimintoihin[37]:

- Se on vapaa.
- Avoimen lähdekoodin periaatteiden mukaisesti koodia voi muokata haluamakseen.
- Avoimen lähdekoodin ja avointen standardien käyttö ei sido käyttäjiä määrättyihin toimintatapoihin.
- Toimii Jakarta Tomcatin alaisuudessa.
- Helppo asentaa ja ylläpitää.
- Käyttää SOAPia (Simple Object Access Protocol) asiakkaan ja palvelimen väliseen kommunikaatioon.
- Graafinen käyttöliittymä toimii jokaisessa Javaa tukevassa käyttöjärjestelmässä. Testattu erilaisissa Windows- ja Linux-ympäristöissä.
- Sisäänrakennettu visuaalinen muutostenesitystyökalu.
- Graafisen käyttöliittymän lokalisointi on helppoa.
- Ohjelma kehittyy koko ajan. Uusia parannuksia on suunnitteilla lähitulevaisuudessa.

Esitettyjen, lievästi pintapuolisten, ominaisuuksien lisäksi SourceJammer tarjoaa myös perustason versionhallintatoiminnot.

[37] http://www.sourcejammer.org/usersguide21/whySJ.html (viitattu 9.10.2007).

4.11.2 Toiminta

SourceJammerissa palvelinpään tehtävänä on hallita tiedostoja ja versiohistoriaa. Se huolehtii versionhallinnan perustoiminnoista, kuten versioiden hakemisesta tietovarastosta muokattavaksi ja uuden version tallentamisesta takaisin tietovarastoon. Asiakaspään tehtävänä puolestaan on kommunikoida palvelinpään kanssa ja huolehtia tietovarastosta haetuista tiedostoista.

Palvelin järjestää kaikki tiedostot *arkistoiksi* (engl. archives). Yhdellä palvelimella voi olla useita arkistoja, mutta niihin pääsee käsiksi vain sopivien oikeuksien avulla. Tuetut oikeustasot SourceJammerin versiosta 1.2 lähtien ovat seuraavat:

Käyttäjä (engl. user): Kaikki käyttäjät.

Ylläpitäjä (engl. admin): Ylläpitäjällä on kaikki oikeudet kaikkiin arkistoihin.

Arkistonhoitaja (engl. archive controller): Ylläpitäjä voi määrätä yksittäisille arkistoille arkistonhoitajan, jolle annetaan ylläpitäjän oikeudet kyseisiin arkistoihin.

Arkistonkäyttäjä (engl. archive user): Käyttäjä, joka voi käyttää tietyn arkiston perustoimintoja, kuten lisätä ja poistaa tiedostoja, tehdä muutoksia ym.

Tuntematon (engl. anonymous): Tuntematon käyttäjä voi lukea arkiston tietoja, mikäli tämä oikeus on sallittu.

SourceJammerissa on mahdollisuus liittää arkiston sisällön tietyllä hetkellä vallitsevaan tilaan symbolinen tunniste (engl. label), jolla siihen voidaan myöhemmin viitata. Nimeäminen luo xml-tiedoston, joka sisältää tiedot kaikista arkistossa annetulla hetkellä olevista tiedostoista, niiden sijainneista ja versioista. Annetun tunnisteen avulla tähän merkittyyn tilanteeseen voidaan palata milloin tahansa, riippumatta siitä, onko tiedostoja myöhemmin muutettu. SourceJammerissa tästä ominaisuudesta käytetään nimeä *Label*, mutta toiminnaltaan se vastaa muiden versionhallintaohjelmien yleisemmin käyttämää käsitettä *tag*.

Arkiston sisällä tiedosto voidaan jakaa usean hakemiston kesken siten, että sama tiedosto näkyy eri hakemistoissa, ja siihen tehdyt muutokset päivittyvät automaattisesti jokaiseen paikkaan. Tämä toiminto vastaa Unix-järjestelmien kovien linkkien käyttöä myös siinä yksityiskohdassa, että vain tiedostoja voi jakaa. Hakemiston jakaminen tehdään jakamalla kaikki sen sisältämät tiedostot erikseen, mihin SourceJammer tarjoaa oman toiminnon.

Jaetut tiedostot voivat haarautua siten, että haarautuva tiedosto irtautuu muista jaoista, mutta säilyttää yhteisen versiohistorian. Muut jaot jäävät ennalleen, mutta uuteen haaraan tehdyt muutokset eivät enää näy niissä.

SourceJammer tukee tekstitiedostojen kanssa avainsanojen laajentamista (engl. keyword expansion). Toiminto korvaa tiedoston sisällä merkityt avainsanat sopivilla arvoilla aina tarvittaessa. Esimerkiksi tiedoston `main.c` sisältämä avainsana `\$FileName: \$` korvataan arvolla `\$FileName: main.c\$`. Turvallisuuden vuoksi järjestelmälle on kerrottava kaikki tiedostopäätteet, joihin laajennosta voidaan soveltaa.

Palvelin säilyttää kaikki arkistot konfiguraatiotiedoston määrittämässä hakemistossa. Jokainen arkisto tallennetaan kyseisen hakemiston alle oman nimensä mukaiseen alihakemistoon.

Jokainen arkistohakemisto sisältää seuraavat alihakemistot:

file sisältää tietoja tiedostoista xml-muodossa.

label sisältää tietoja nimetyistä tiloista xml-muodossa.

project sisältää tietoja projekteista xml-muodossa.

source sisältää palvelimelle tallennetut lähdekooditiedostot joko kokonaisina tai deltoina.

version sisältää versioihin liittyvät kommentit tekstimuodossa.

Tiedostoja kuvaavat xml-tiedostot sisältävät tiedot eri tiedostoversioista, tiedostoon liittyvistä kommenttitiedostoista ja varsinaisista lähdekooditiedostoista. Projektitiedostot sisältävät tiedot kaikista projektiin kuuluvista tiedostoista ja aliprojekteista, sekä viitteet tiedostojen xml-kuvauksiin.

4.11.3 Nykytila

SourceJammerin kehitys on melko hidasta. Viimeisin julkaistu versio on 2.1.2 beta maaliskuulta 2005. Kahden ja puolen vuoden odotusaika ei anna kovin lupaavaa kuvaa ohjelman tulevaisuudesta, ja onkin epätodennäköistä, että SourceJammer pystyy kilpailemaan uusien versionhallintaohjelmien kanssa. Graafisuus voi auttaa jonkin verran, mutta yleisten kehitysympäristöjen, kuten Eclipsen, tarjoama tuki uusille versionhallintaohjelmille kuroo tämänkin etumatkan umpeen.

4.12 Arch

GNU Arch[38] on Tom Lordin kehittämä revisioiden, lähdekoodin ja konfiguraatioiden hallintaan tarkoitettu työkalu. Sovelluksen www-sivustolla nämä osa-alueet määritellään seuraavasti[39]:

Revisionhallintajärjestelmä (engl. a revision control system) on puumuotoiseen hierarkiaan tallennettuja tiedostoja ja niihin tehtävä muutoksia koordinoiva työkalu. Tyypillisissä ohjelmistoprojekteissa revisionhallintajärjestelmää käytetään lähdekoodin tilan ja koodiin tehtyjen muutosten, kuten bugikorjausten ja uusien ominaisuuksien, seurantaan. Sen avulla kaikkien kehittäjien tekemät muutokset voidaan synkronoida, ja eri aikoina tehdyt muutokset voidaan yhdistää yhdeksi kokonaisuudeksi.

Lähdekoodinhallintatyökalu (engl. a source management tool) auttaa hallitsemaan suuria tiedostomääriä. Sen avulla voidaan selvittää mm. kaikki työhakemistossa olevat lähdekoodi- ja objektitiedostot, ja selvittää, mitä tiedostoja on lisätty tai poistettu.

Konfiguraationhallinta (engl. configuration management) auttaa useiden erikseen hallittujen lähdekoodien yhdistämisessä yhdeksi kokonaisuudeksi. Työkalu auttaa yhdistetyn projektin muodostamisessa ja hallinnoi erillisten komponenttien synkronointia.

Arch on GPL-lisenssin alainen vapaa ohjelma. Sen www-sivustoa ei ole päivitetty pitkään aikaan, joten siellä on jonkin verran vanhentunutta ja siten paikkansapitämätöntä tietoa. Siellä mm. väitetään, että Linux-ytimen pääkehittäjät käyttäisivät epävapaata versionhallintajärjestelmää, mikä viittaa BitKeeperiin. Sen käyttö kuitenkin lopetettiin jo vuoden 2005 alussa.

Sivustonsa mukaan Archin yhtenä tavoitteena on kehitysprojektien parantaminen. Siirtymisen CVS:stä Archiin sanotaan tekevän kehityksestä tehokkaampaa.

4.12.1 Ominaisuudet

Archia alettiin kehittää olosuhteissa, joissa hajautettu kehitys oli jo osoittautunut toimivaksi, mutta vapaat työkalut eivät vielä tarjonneet siihen kunnollista tukea. Tästä syystä Arch kiinnitti huomiota juuri hajautetun kehityksen vaatimuksiin:

[38] http://www.gnu.org/software/gnu-arch/ (viitattu 1.8.2007).

[39] www.gnu.org/software/gnu-arch/tutorial/Introducing-arch.html (viitattu 10.10.2007).

Käsittelee kokonaisia hakemistoja (engl. works on whole trees): Arch käsittelee kerralla kokonaisia hakemistoja. Usean tiedoston muutokset voidaan merkitä yhdeksi kokonaisuudeksi. Tiedostojen siirto hakemistosta toiseen voidaan merkitä muiden muutosten rinnalle omaksi muutokseksi.

Muutosjoukkokeskeinen (engl. changeset oriented): Arch liittää jokaisen tiedostoversion johonkin muutosjoukkoon. Muutosjoukko sisältää tiedot siitä, mitä muutos tekee. Muutosten ja muutoshistorian tarkasteluun on omat muutosjoukko-orientoituneet komentonsa. Muutosten yhdistämistoiminto osaa selvittää, mitä muutosjoukkoja on jo yhdistetty.

Täysin hajautettu (engl. fully distributed): Arch ei käytä keskitettyä tietovarastoa. Jokaisella kehittäjällä voi olla oma tietovarasto, jonne muutokset tallennetaan. Archin komennot osaavat toimia useiden tietovarastojen kanssa saumattomasti.

Esitellyt ominaisuudet ovat nykyään modernien versionhallintaohjelmien perusominaisuuksia, mutta Arch oli ensimmäinen vapaa ohjelma, jossa niitä käytettiin.

GNU Arch -spesifikaatiolle on useita toteutuksia. Alkuperäistä larch-toteutusta seurasivat mm. tla[40] ja ArX, joista ensimmäisestä erkani myöhemmin Bazaar 1.x (baz), ja joista jälkimmäinen eriytyi kokonaan omaksi ohjelmaksi. Yhteisen spesifikaation ansiosta jokainen näistä pystyi kuitenkin käyttämään samaa tiedostorakennetta, ja kun baz myöhemmin sai oman tiedostorakenteen, myös tla oppi käyttämään sitä.

4.12.2 Toiminta

Archissa tietovarastoa nimitetään arkistoksi (engl. archive). Muista versionhallintaohjelmista poiketen siinä on varsin tiukat nimeämiskäytännöt, jotka tekevät siitä hankalasti opittavan. Esimerkiksi arkiston nimen on oltava muotoa sähköpostiosoite, kaksi viivaa ja loppuosa, jolloin hyväksyttävä nimi voi näyttää esimerkiksi seuraavalta:

```
matti.meikalainen@yritys.fi--2006-projekti
```

Uusi arkisto luodaan komennolla `tla make-archive`, jolle annetaan tiedot sen nimestä ja sijainnista. Luotu arkisto voidaan haluttaessa määritellä oletusarkistoksi,

[40] Tom Lordin mukaan tla on lyhenne, joka tulee sanoista “true love, always” tai “three letter acronym”. http://lists.gnu.org/archive/html/gnu-arch-users/2005-12/msg00128.html

jolloin sitä käytetään kaikissa niissä tapauksissa, joissa käytettävää arkistoa ei erikseen kerrota.

Arkistoissa lähdekoodit jaetaan kategorioihin, kategoriat haaroihin ja haarat lopulta versioihin. Uuden projektin aloittamisen yhteydessä järjestelmälle on annettava nämä kaikki tiedot, joten esimerkiksi yksinkertainen "hello world"-ohjelmaprojekti voidaan perustaa komennolla `tla archive-setup hello-world-main-0.1`. Annettu komento luo kategorian nimeltä *hello-world*, ja aloittaa siellä *main*-nimisen kehityshaaran versiosta *0.1* alkaen. Monimutkainen nimeämisjärjestelmä selittyy sillä, että mallia on otettu kirjastojen käyttämästä Deweyn luokittelujärjestelmästä.

Lähdekoodihakemisto voidaan muuttaa versioiduksi työhakemistoksi komennolla `tla init-tree hello-world-main-0.1`, missä *init-tree*:n jälkeinen osa nimeää projektin, johon työhakemisto liittyy. Komento lisää hakemistoon uuden alihakemiston *{arch}*, jonne metadatan tallennetaan. Arch käyttää tiedostonimissä myös muita erikoisia merkkejä, kuten +-, =- ja ,-merkkejä, mikä voi aiheuttaa joskus ongelmia joidenkin työkalujen kohdalla.

Lähdekoodit laitetaan versionhallintaan komennolla `tla add`. Jokaisella versioidulla tiedostolla on käytössä sekä *polkunimi* että *inventaarionimi* (engl. inventory id). Polkunimi kuvaa tiedoston suhteellisen sijainnin työhakemiston juuren suhteen, ja inventaarionimi on tiedostolle annettu yksilöllinen tunniste, joka säilyy riippumatta siitä, onko tiedosto uudelleennimetty tai siirretty eri paikkaan.

Muista versionhallintasovelluksista poiketen Archissa muutosloki suositellaan kirjoitettavaksi jo ennen muutosten tekoa, jolloin käyttäjä ei harhaudu tekemään mitään asiaankuulumatonta. Valmiit muutokset tallennetaan arkistoon uudeksi versioksi komennolla `tla commit`, mutta niiden sisältöä ja vastaavuutta kirjoitettuun lokitekstiin voi tarkastella jo ennen tallennusta komennolla `tla changes`.

Arkistossa alkuperäiset tiedostot pidetään pakatussa tar-tiedostossa, mutta jokaista uutta versiota varten luodaan erillinen alihakemisto. Hakemistoon laitetaan muutoslokin lisäksi *Change Set*-tiedosto, joka sisältää tehdyt muutokset pakattuna diff-tiedostona.

Vanhat versiot saa ulos arkistosta `tla get` -komennolla. Komento purkaa ensin kokonaisen hakemistopuun sisältävän tar-paketin, minkä jälkeen se soveltaa (engl. apply) purettuun hakemistoon muutostiedostojen sisältämiä diff-tiedostoja, kunnes tuloksena on haluttu versio. Diff-tiedostojen soveltaminen hidastaa tietojen ulossaantia silloin, kun muutoksia on paljon. Tähän Arch tarjoaa ratkaisuksi välimuistin käyttöä, mikä tarkoittaa yksinkertaisesti sitä, että tietyt versiot tallennetaan arkis-

toon kokonaisina, jolloin niiden hakuun riittää tiedostojen kopiointi oikeaan paikkaan.

Arch tarjoaa muutosten yhdistämiseen useita eri komentoja, joiden kaikkien taustalla on näkemys, että yhdistäminen on loppujen lopuksi muutostiedostojen tekoa ja soveltamista[41].

Komento `tla replay` laskee ensin annetun revision ja sen esi-isän väliset muutokset ja laittaa ne työhakemistoon. Sama komento voi myös soveltaa useita muutoksia sen perusteella, löytyykö niitä vastaavia lokeja työhakemistosta. Komento `tla update` toimii samalla tavalla, mutta laskee deltan työhakemiston sisältämän uusimman version ja annetun tietovaraston uusimman version väliltä.

Update-komento toimii tilanteessa, jossa kehittäjä A päivittää työhakemistoonsa B:n tekemät muutokset. Ongelmia tulee silloin, jos B päivittää itselleen A:n tekemät muutokset. Tällöin nimittäin käy niin, että A:n seuraavan päivityksen yhteydessä komento näkee B:llä uusia muutoksia, vaikka ne ovatkin peräisin A:lta. Yhdistämisyritys johtaa konfliktiin.

Tämän tilanteen korjaamiseksi Arch tarjoaa komennon `tla star-merge`. Komento toimii kuten `tla update`, mutta muutoksia selvittäessään etsii ensin viimeisimmän yhteisen version, jolloin käsiteltäviksi muutoksiksi lasketaan muutokset yhteisestä versiosta tietovaraston versioon. Haluttaessa voidaan myös tehdä 3-way-merge yhteisen version, tietovaraston version ja työhakemiston version välillä.

4.12.3 Ongelmia

Vaikka Arch on käyttökelpoinen sovellus, siinä on tai on ollut joitakin ongelmia, jotka ovat heikentäneet sen käytettävyyttä:

Arch voi olla hidas Yhdessä tapauksessa paikallisen Arch-tietovaraston käyttö oli hitaampaa, kuin verkossa olevan CVS-tietovaraston käyttö[42]. Revisiokirjaston käyttö auttaa tähän ongelmaan jonkin verran.

Binääritiedostot tallennetaan useaan kertaan Archin käyttäminen patch-tiedostojen on oltava kääntyviä eli muutos pitää pystyä sekä ottamaan käyttöön että poistamaan. Tämän vuoksi binääritiedostoon tehtyjen muutosten muutostiedosto sisältää kokonaiset versiot tiedostosta ennen muutosta ja muutoksen jälkeen[43].

[41] http://www.gnuarch.org/gnuarchwiki/Merging_with_Arch
[42] http://lists.gnu.org/archive/html/gnu-arch-users/2005-03/msg00070.html
[43] http://lists.gnu.org/archive/html/gnu-arch-users/2005-03/msg00270.html

Tiedostonimissä ei saanut olla välejä Vaikka Arch itse käyttää erikoismerkkejä omissa tiedostonimissään, se ei ole aina osannut käsitellä tiedostoja, joiden nimissä on välilyöntejä. Tämä ongelma korjattiin vasta versiossa 1.2.1[44].

Yhdistämistoiminto ei osaa käsitellä identtisiä muutoksia Käytetty star-merge ei osaa käsitellä tilannetta, missä kahdessa eri paikassa on tehty samanlaiset muutokset[45].

Ongelmiin on keksitty monia kiertoteitä, mutta vähintään yhdistämistoiminnon ongelma jarruttaa Archin käyttöä kovin hajautetussa kehityksessä.

4.12.4 BitKeeper-kriisin vaikutus Archiin

Helmikuussa 2005 listalla oli keskustelua tietojen siirtämisestä versionhallintaohjelmasta toiseen. Catalin Marinas oli tehnyt skriptin, jolla Linux-ytimen tiedot voitiin saada BitKeeper-tietovarastosta Archiin käyttämällä BitMoverin BK-CVS-palvelua. Hän totesi, että välissä olevan CVS:n takia tiedostojen uudelleennimeämistiedot hävisivät.

Keskustelun aikana selvisi, että vanhemmissa BitKeeper-muutoksissa tarvittavat tiedot löytyivät alkukommenteista, mutta ne oli uusissa muutoksissa poistettu. Tämän katsottiin olevan mahdollisesti sattumaa, mutta todennäköisempänä pidettiin tietojen tahallista poistoa[46]. Osallistujat pelkäsivät, että kaikki yritykset tietojen selvittämiseksi julkisista lähteistä tulisivat epäonnistumaan. Myös joidenkin Linux-kehittäjien asenteita ihmeteltiin[47].

Kun BitMover ilmoitti huhtikuussa 2005 lopettavansa BitKeeper-ohjelman ilmaisversion[48] , ja Linus ilmoitti heti seuraavana päivänä etsivänsä vaihtoehtoja BitKeeperille[49] , Arch-kehittäjät tarttuivat heti mahdollisuuteen. Tom Lord kirjoitti Linusille avoimen sähköpostiviestin, jossa hän kuvasi Archin perusideoita. Hänen epäonnekseen Linus oli jo hieman aiemmin aloitellut oman versionhallintaohjelman toteuttamista. Lordin viestiä edeltävänä päivänä Linusin ohjelma oli jo siinä vaiheessa, että se pystyi säilyttämään oman historiansa (engl. self hosting).

[44] http://www.gnuarch.org/gnuarchwiki/SubVersionAndCvsComparison
[45] http://lists.gnu.org/archive/html/gnu-arch-users/2005-04/msg00246.html
[46] http://lists.gnu.org/archive/html/gnu-arch-users/2005-02/msg00195.html
[47] http://lists.gnu.org/archive/html/gnu-arch-users/2005-02/msg00197.html
[48] http://www.bitkeeper.com/press/2005-04-05.html (viitattu 1.10.2007).
[49] http://lists.gnu.org/archive/html/gnu-arch-users/2005-04/msg00068.html

Tom Lord tutustui GITin ideoihin ja piti niistä niin paljon, että jo parin viikon sisällä hän ilmoitti Archin "adoptoivan" GITin[50]. Hän täsmensi tätä kertomalla, miten GITiä käytettäisiin uuden revisiokirjaston/arkiston perustana. Arch toteuttaisi varsinaisen versionhallintaosuuden.

Jotkut GITin ominaisuudet herättivät keskustelua. Esimerkiksi tapaa laskea tiivistearvo pakatusta binääriobjektista (engl. blob, binary large object) ihmeteltiin, sillä pakkausalgoritmeja ei pidetty deterministisinä[51]. Tapa myös vaikeutti eri pakkausalgoritmien kokeilua. Myöhemmin tiivistearvon laskenta siirrettiin edeltämään objektin pakkausta[52].

Joidenkin mielestä Linusin tapa käyttää tiivistefunktioita ei ollut hyvä asia[53]. Hajautusfunktioiden kanssa voi tulla törmäyksiä, eli eri objektit voivat saada saman tiivistearvon. Tämä ei sovi versionhallintaohjelmalle, sillä se saattaa hävittää tietoa huomaamatta. Tämän mahdollisuuden arveltiin kuitenkin olevan hyvin epätodennäköinen[54].

Myös tiivistefunktioiden vaikutusta nopeuteen kritisoitiin. Tavallinen hajautustaulukko on tunnetusti nopea, mutta GITin tapauksessa käytetyn tiivistefunktion arvoalue oli niin iso, ettei hajautustaulukko mahtuisi muistiin. Vaihtoehtona esitettiin B-puun[55] käyttöä.

Eräs keskusteluun osallistuja piti GITiä Linusin "paniikkiratkaisuna"[56].

Kritiikistä huolimatta Tom Lord jatkoi GIT-vaikutteisen Arch-version kehittämistä. Heinäkuussa 2005 hän julkaisi ensimmäisen version revc-nimisestä ohjelmasta, nimittäen sitä GNU Arch 2.0:ksi[57]. Uusi ohjelma poikkesi monelta osin aikaisemmasta Arch 1.x -sarjasta:

- Komentojen määrä oli supistettu kymmeneen.
- Erikoismerkkejä sisältävät tiedosto- ja hakemistonimet oli korvattu yksinkertaisella `.revc`-hakemistolla, jonka sisältöä ei tarvinnut muokata käsin.
- Revisioiden nimeämistapa oli huomattavasti aiempaa vapaampi. Vain kauttaviivan ("/") käyttö oli kielletty ja nimien maksimipituutta hieman rajoitettu.

[50] http://lists.gnu.org/archive/html/gnu-arch-users/2005-04/msg00176.html
[51] http://lists.gnu.org/archive/html/gnu-arch-users/2005-04/msg00172.html
[52] http://lists.gnu.org/archive/html/gnu-arch-users/2005-04/msg00185.html
[53] http://lists.gnu.org/archive/html/gnu-arch-users/2005-04/msg00221.html
[54] http://lists.gnu.org/archive/html/gnu-arch-users/2005-04/msg00235.html
[55] Eräänlainen tasapainotettu puurakenne.
[56] http://lists.gnu.org/archive/html/gnu-arch-users/2005-04/msg00237.html
[57] http://lists.gnu.org/archive/html/gnu-arch-users/2005-07/msg00023.html

- Arkistojen käyttöä oli yksinkertaistettu. Arkistot olivat nimettömiä ja niitä voitiin peilata peruskomentojen, kuten rsyncin, avulla.
- Vaikka koodia ei oltu optimoitu, ohjelma oli huomattavan nopea.

Lord näytti hyväksyneen GITin periaatteet täysin. Hän mm. puolusti revc:n (ja GITin) tapaa tallentaa muuttuneet tiedostot tilaa säästävien deltojen sijaan kokonaisina. Hänen mukaansa modernit versionhallintaohjelmat voivat täysin vapaasti käyttää jopa tuhatkertaisesti enemmän tallennustilaa kuin CVS, mikäli se antaa käyttäjälle paremman käyttökokemuksen[58].

Uusi ohjelma nostatti ristiriitaisia tunteita. Vaikka se saattoikin olla edistysaskel Archille, miksi ylipäätään kannatti tehdä täysin uusi ohjelma, jos GIT oli niin hyvä? Näistä arveluista huolimatta Lordin innostus pysyi yllä jonkin aikaa, mutta elokuussa hän lähetti listalle tietoja kuolleista projekteistaan, joiden joukkoon revc:kin oli päätynyt[59].

Lopulta henkilökohtaiset ongelmat johtivat siihen, että Lord luopui Archin ylläpitäjyydestä. Tämä johti keskusteluun sovelluksen tulevaisuudesta, jolloin Lord ehdotti, että Bazaar 1.x sarja nimettäisiin viralliseksi GNU Arch -versioksi[60]. Keskustelussa todettiin, että suurin osa kehittäjistä oli jo siirtynytkin Bazaarin käyttäjiksi, mutta Martin Langhoff puolestaan totesi aikovansa siirtyä GITin (oikeastaan cogiton) käyttäjäksi[61]. Myöhemmin hän ilmoitti tehneensä skriptin, jolla tiedot voitiin siirtää Archista GITiin[62].

Vaikka tla:n kehitys oli hidastunut, sitä pidettiin vielä täysin käyttökelpoisena vaihtoehtona pieniin projekteihin. Lokakuussa 2005 Archin uudeksi ylläpitäjäksi hyväksyttiin Andy Tai, joka ilmoitti tavoitteekseen pitää sovellus käyttökelpoisena siihen asti, kunnes uuden sukupolven versionhallintaohjelmien joukosta nousisi selvä voittaja. Samalla hän totesi, että uudet Arch-julkaisut tulevat perustumaan tla 1.3.x -sarjaan[63].

Uuden ylläpitäjän aloitettua into tla:n kehittämiseen taas kasvoi. Ludovic Courtès päivitti Tom Lordin tekemän tutoriaalin[64] ja Lode Leroy lisäsi sovellukseen tuen

[58] http://lists.gnu.org/archive/html/gnu-arch-users/2005-07/msg00041.html
[59] http://lists.gnu.org/archive/html/gnu-arch-users/2005-08/msg00131.html
[60] http://lists.gnu.org/archive/html/gnu-arch-users/2005-08/msg00030.html
[61] http://lists.gnu.org/archive/html/gnu-arch-users/2005-08/msg00042.html
[62] http://lists.gnu.org/archive/html/gnu-arch-users/2005-08/msg00147.html
[63] http://lists.gnu.org/archive/html/gnu-arch-users/2005-10/msg00246.html
[64] http://lists.gnu.org/archive/html/gnu-arch-users/2006-02/msg00038.html

Cygwin-alustalle[65].

Myös Tom Lord muuttui jälleen aktiiviseksi ja aloitti useita keskusteluita eri aiheista. Hänen mielenkiintonsa revc:hen oli palannut, mutta muut keskittyivät nyt vain vanhemman tla:n kehittämiseen. Vuonna 2006 Googlen *Summer of Code* -ohjelma johti kuitenkin laajaan keskusteluun Arch 2.0:n tulevaisuudesta, ja Lord alkoi kerätä mielipiteitä ohjelman kohderyhmästä ja tavoitteista[66]. Tilanne johti siihen, että Saurabh Sehgal alkoi toteuttaa Arch 2.0:n puuttuvia ominaisuuksia SoC-projektin alaisuudessa[67] , mutta projekti ei kuitenkaan päässyt loppuun asti[68]. Epäonnistumisesta johtuen mielenkiinto uutta sovellusta kohtaan hävisi, ja voimavarat keskitettiin jälleen vanhan tla:n kehitykseen.

4.12.5 Nykytila

Syyskuussa 2007 sähköpostilistalla keskusteltiin uuden GPLv3-lisenssin käyttämisestä. Kukaan ei varsinaisesti vastustanut lisenssin päivittämistä, mutta yleistä keskustelua sen tarpeellisuudesta syntyi jonkin verran.

Nykyään tla on käytössä mm. Savannah-projektisivustolla[69], mutta sen käyttäjämäärä ei näytä enää olevan kovin suuri. Bazaar-NG (bzr) on ottanut alkuperäisen Bazaarin (baz) paikan (ja Bazaar-nimen), eikä Arch 2.0 näytä edistyneen vuoden 2006 epäonnistuneen kehitysprojektin jälkeen.

Vaikka tla onkin jo vanha ohjelma, sitä käytetään jonkin verran sellaisissa projekteissa, joissa se on ollut käytössä jo aiemmin. Uusiin projekteihin sitä ei enää kannata käyttää.

4.13 DCVS

DCVS (Distributed CVS)[70] on elego Software Solutions GmbH:n kehittämä hajautettu versionhallintajärjestelmä. Nimensä mukaisesti se perustuu CVS:ään ja lisää siihen joitakin hajautetun kehityksen vaatimia ominaisuuksia.

DCVS:n käyttämät tietovarastot ovat suoraan yhteensopivia CVS:n käyttämien

[65] http://lists.gnu.org/archive/html/gnu-arch-users/2006-03/msg00022.html
[66] http://lists.gnu.org/archive/html/gnu-arch-users/2006-04/msg00143.html
[67] http://lists.gnu.org/archive/html/gnu-arch-users/2006-05/msg00062.html
[68] http://lists.gnu.org/archive/html/gnu-arch-users/2006-07/msg00020.html
[69] http://savannah.gnu.org (viitattu 18.10.2007).
[70] http://dcvs.elegosoft.com/ (viitattu 10.8.2007).

kanssa, joten niitä voidaan käyttää myös sen asiakasohjelmilla. Hajautuksen DCVS toteuttaa myöhemmin esiteltävän ns. replikointiverkon avulla.

4.13.1 Ominaisuudet

DCVS perustuu alkuperäiseen CVS:n lähdekoodiin, ja se on toteutettu niin, että se voidaan myöhemmin yhdistää viralliseen CVS:ään. Tämän vuoksi näiden sovellusten välinen yhteensopivuus on erittäin hyvä, mikä on ollut merkittävä etu ainakin CVS:n valtakaudella.

DCVS ei juurikaan muuta CVS:n toimintoja, vaan ainoastaan lisää siihen seuraavia ominaisuuksia, jotka ovat tarpeellisia tai hyödyllisiä hajautetussa kehityksessä:

- Usean samanaikaisen kehittäjän tuki. Jokainen kehittäjä voi muokata koodia toisista riippumatta, koska työtilat on eristetty toisistaan.
- Tietovarastojen levitys maailmanlaajuisesti turvallisen yhteyden läpi.
- Tehokas tiedon replikointi ja nopeat toiminnot paikallisen palvelimen kanssa.
- CVS-protokollan käyttö asiakkaan ja palvelimen välillä. Tämä mahdollistaa integroinnin useimpiin kehitysympäristöihin, koska CVS on yleisesti tuettu järjestelmä.
- Perinteisten tagien korvaus tilannekuvilla (engl. snapshot).
- Change Set -toiminnallisuus yksittäisten tiedostojen versioinnin sijaan.
- Sähköpostitse lähetettävät muistutukset.
- Käyttäjienhallinta ja autentikointi joko paikallisten asetusten mukaisesti ja hajautettujen LDAP-palvelinten kautta.
- Asennuspaketit kaikille yleisimmille käyttöjärjestelmille.

DCVS on vapaa ohjelma. Se koostuu kahdesta osasta, jotka ovat eri lisenssien alla. CVS-ohjelmaa käyttävä osa on lisensoitu GPL:n alaisuuteen ja CVSup-ohjelmaa käyttävä osa BSD-tyyppisen lisenssin alaisuuteen.

DCVS on muutosjoukkopohjainen, eli se käsittelee usean tiedoston muutoksia kokonaisuutena. Changeset-tiedostot sisältävät muutosten lisäksi tietoa mm. luontiajankohdastaan, tekijästään ja käytetystä luontikomennosta.

CVS:n tagien lisäksi DCVS tukee tilannekuvia eli *snapshot*eja. Snapshot-tiedostot ovat tekstitiedostoja, jotka sisältävät tietoja mm. jokaisen tiedoston senhetkisestä versiosta. Niiden avulla voidaan tallentaa erilaisia tiedostokonfiguraatioita, jotka voidaan myöhemmin saada nopeasti käyttöön.

4.13.2 Toiminta

Hajautukseen DCVS käyttää ns. *replikointiverkkoa*, missä jokaiselle tietovarastoja sisältävälle palvelimelle varataan jokin erillinen lukualue, esimerkiksi 1–1000 ja 1001–2000. Tämän jälkeen jokainen kehityshaara liitetään johonkin tiettyyn palvelimeen siten, että sille annetaan tunnisteeksi jokin luku palvelimelle varatulta lukualueelta.

Kehityshaaroja voidaan vapaasti replikoida eri palvelinten välillä, mutta niiden muokkaaminen on rajoitettu vain siihen palvelimeen, mihin kyseinen kehityshaara on liitetty. Käytännössä tämä tarkoittaa sitä, että tietoja voi ottaa ulos miltä tahansa palvelimelta, mutta niitä voi muokata vain silloin, jos käsiteltävän kehityshaaran tunniste löytyy käytetyn palvelimen lukualueelta. Tämän rajoituksen tarkoituksena on vähentää samanaikaisen kehityksen aiheuttamia konflikteja.

Mikäli kehittäjä muuttaa replikoidusta tietovarastosta haettua koodia, hän ei voi tallentaa tekemiään muutoksia suoraan, vaan hänen on luotava niitä varten käyttämälleen palvelimelle uusi kehityshaara. Myöhemmin alkuperäistä tietovarastoa käyttävä kehittäjä voi yhdistää uuden kehityshaaran muutokset, jolloin muutokset leviävät ajan myötä myös muihin kopioihin.

Replikointiverkossa tietovarastojen sisällöt päivittyvät automaattisesti tietyin väliajoin. Päivitys ei tapahdu kaikkiin palvelimiin samanaikaisesti, mutta käytännössä tämä ei aiheuta ongelmia. Replikointi tapahtuu yksinkertaisesti siten, että palvelin päivittää kopionsa ulkoisen *cvsup*-ohjelman avulla.

4.13.3 Nykytila

DCVS:n parhaana puolena aikaisempina vuosina oli sen yhteentoimivuus CVS-asiakasohjelmien kanssa, mutta CVS:n aseman hiivuttua tämä ei tuo enää lisäarvoa. Vaikka DCVS on käyttökelpoinen vielä nykyäänkin, sitä ei enää suositella uusille käyttäjille, joita kehotetaan siirtymään suoraan Subversioniin. Mikäli yhteensopivuudesta CVS:n kanssa ei voi tinkiä, DCVS:n käyttäminen CVS:n tilalla tuo kuitenkin jonkin verran parannuksia mm. hajautettuun kehitykseen.

DCVS:n viimeisin julkaistu versio on 1.0.3 syyskuulta 2006[71].

4.14 Meta-CVS

Meta-CVS[72] on Kaz Kylhekun CVS:n päälle rakentama versionhallintaohjelma, joka säilyttää kaikki CVS:n toiminnot, mutta tarjoaa niiden lisäksi uusia ominaisuuksia. Se on helppokäyttöisempi kuin CVS, mutta korjaa samalla CVS:n suurimpia ongelmia.

Meta-CVS:n kehitys alkoi tammikuussa 2002[73]. Kylheku oli odottanut CVS:n seuraajana tunnetun Subversionin valmistumista, mutta päästyään kokeilemaan sitä totesi, ettei se vastannut hänen odotuksiaan.

4.14.1 Ominaisuudet

Meta-CVS tarjoaa CVS:n toimintojen lisäksi mm. hakemistojen versioinnin. Ohjelma yksinkertaistaa myös kehityshaarojen käyttöä, muistamalla miten muutoksia on yhdistetty haarasta toiseen.

Ohjelman pääominaisuuksia ovat:

Hakemistojen versiointi Hakemistot versioidaan samalla tavalla kuin tiedostot, joten myös niitä voidaan muokata eri kehityshaaroissa ja yhdistää muutokset jälkeenpäin.

Tiedostotyyppien käsittely Järjestelmä osaa tunnistaa, mitkä tiedostot ovat binäärisiä ja mitkä tekstimuotoisia. CVS:ssä tämä tieto piti antaa komennolle erikseen.

Erikoistapausten käsittely Järjestelmä osaa käsitellä CVS:lle vaikeita tapauksia, kuten esimerkiksi välilyöntejä sisältäviä tiedostonimiä.

Yksinkertaiset haarautumis- ja yhdistämistoiminnot Kehityshaarojen luonti ja yhdistäminen onnistuvat yksinkertaisilla komennoilla. Järjestelmä muistaa, mitä haaroja on yhdistetty minnekin.

[71] http://dcvs.elegosoft.com/cgi-bin/cvsweb.cgi/~checkout~/doc/ANNOUNCE-1.0.3.EN?rev=1.3;content-type=text%2Fplain;cvsroot=DCVS (viitattu 22.10.2007).

[72] http://users.footprints.net/~kaz/mcvs.html (viitattu 1.8.2007).

[73] http://users.footprints.net/~kaz/mcvs-criticisms-2.html

Symbolisten linkkien käsittely Järjestelmä osaa versioida myös symboliset linkit.

Tuki metadatalle Jokaiseen tiedostoon voidaan liittää versioitavaa metadataa. Esimerkiksi tiedostoihin liittyy oletuksena tieto siitä, ovatko ne ajettavia.

Ulkoisen koodin seuranta Järjestelmä tukee ns. *grab*-toimintoa. Sen avulla järjestelmän ulkopuolella kehitetty koodi haetaan erilliseen kehityshaaraan. Tämän jälkeen järjestelmä analysoi koodin, tunnistaen mm. tiedostojen lisäykset, poistot ja uudelleennimeämiset.

Helppo käyttöönotto Koska järjestelmä toimii tavallisen CVS-palvelimen kanssa, se ei vaadi muutoksia palvelinpuolelle.

Meta-CVS on toteutettu Common Lisp -ohjelmointikielellä [2] ja se käyttää kyseisen kielen GNU CLISP -toteutusta[74]. Ohjelman lisenssinä on GPL. Tämä ei kuitenkaan selviä ohjelman kotisivulta, vaan sen lähdekoodipaketista.

4.14.2 Toiminta

Meta-CVS tallentaa tiedostojen eri versiot tietovarastoon mm. RCS:n ja CVS:n tukemassa ,*v*-tiedostomuodossa. Näiden tiedostojen nimet eivät kuitenkaan normaalista poiketen perustu suoraan versioitavien tiedostojen nimiin, vaan ne luodaan satunnaisesti. Tietovaraston sisällä ei myöskään noudateta samanlaista hakemistohierarkiaa kuin työhakemistossa, sillä siellä kaikki tiedostot pidetään samassa hakemistossa[75].

Edellä mainittujen seikkojen vuoksi Meta-CVS ylläpitää normaalien versiotietojen lisäksi erillistä *MAP*-tiedostoa, jonka avulla satunnaisilla tunnisteilla nimetyt versiotiedot osataan kuvata (engl. map) työhakemistoon oikeille paikoilleen. MAP-tiedosto sisältää näiden kuvausten lisäksi myös symbolisten linkkien käsittelyyn tarvittavia tietoja sekä käyttäjän määrittelemiä ominaisuuksia. Toteutuskielensä mukaisesti sen tiedostorakenne on Lisp-syntaksin mukainen. Meta-CVS käyttää työhakemiston juuressa olevaa *MCVS*-hakemistoa metadatan säilyttämiseen.

Tiedosto laitetaan versionhallintaan komennolla `mcvs add`. Toiminto luo tiedostolle satunnaisen 128-bittisen yksilöllisen tunnisteen, jonka heksadesimaaliesi-

[74] http://clisp.cons.org (viitattu 14.8.2007).

[75] http://c2.com/cgi/wiki?MetaCvs (viitattu 25.10.2007).

tystä[76] käytetään tietovarastoon tallennettavan tiedoston nimenä. Samalla MAP-tiedostoon lisätään kuvaus tiedoston oikean nimen ja tämän tunnisteeseen perustuvan nimen välille.

Käytetty tapa suojaa tilanteelta, missä kaksi henkilöä lisää eri tiedostot samalla nimellä samanaikaisesti. Koska tietovarastoon syntyy kaksi erinimistä tiedostoa, kumpikaan tiedosto ei häviä. Konflikti huomataan lisättäessä kuvausta MAP-tiedostoon, jolloin ongelma voidaan ratkaista yksinkertaisella uudelleennimeämisellä.

Muokatut tiedostot tallennetaan tietovarastoon komennolla `mcvs commit`, ja niitä voidaan vapaasti nimetä uudelleen (`mcvs move`). Tiedoston poisto (`mcvs remove`) hävittää MAP-tiedostosta kuvauksen tiedoston ja sen aikaisemman historian väliltä, mutta tiedosto itse jää tietovarastoon. Näin ollen "poisto" voidaan myöhemmin perua (`mcvs restore`), ellei sen historiaa ole ehditty kokonaan hävittää (`mcvs purge`).

Kehityshaarojen käyttö on helppoa. Meta-CVS tarjoaa yksinkertaiset komennot kehityshaarojen luontiin ja yhdistämiseen. CVS:stä poiketen Meta-CVS muistaa, mitä kehityshaaroja on yhdistetty ja milloin. Toiminta on toteutettu CVS:n tavoin merkitsemällä yhdistyskohdat tageilla, mutta Meta-CVS hoitaa tämän kaiken automaattisesti.

Grab-toiminnon avulla versionhallinnan ulkopuolella kehitetty koodi voidaan hakea kehityshaaraan. Meta-CVS analysoi haetun koodin, tunnistaen mm. tiedostojen uudelleennimeämiset, lisäykset ja poistot.

Meta-CVS käyttää työhakemistoista nimeä *hiekkalaatikko* (engl. sandbox). Työhakemistot voivat olla myös sisäkkäisiä, sillä CVS:stä poiketen metadatahakemisto on vain työhakemiston juuressa. Komento `mcvs` tukee sisäkkäisiä työhakemistoja optiolla *–up*, jonka avulla voidaan määrätä, montako työhakemistotasoa ylempänä annettu komento suoritetaan.

4.14.3 Nykytila

Viimeisin julkaistu Meta-CVS on versio 1.0.13 maaliskuulta 2004. Kehitysversio 1.1.0 on julkaistu samaan aikaan.

Koska Meta-CVS pyrkii ainoastaan korjaamaan CVS:n suurimpia puutteita, se ei edes yritä toteuttaa hajautettuun kehitykseen liittyviä toimintoja, mikä saattaa vaikuttaa ohjelman vähäiseen suosioon. ChangeLog-tiedostojen perusteella ohjelmaa

[76] 16-kantainen lukujärjestelmä, missä luvut esitetään merkeillä 0-1 ja a-f. Esim. 0x1a = 16 + 10 = 27 desimaalijärjestelmässä.

kehittää vain yksi henkilö, mikä osoittaa, ettei Meta-CVS ole saanut taakseen kovin suurta käyttäjäkuntaa. Sekä hajautuksen puute että pieni käyttäjäkunta johtavat selvästi siihen lopputulokseen, ettei ohjelmalla ole mahdollisuuksia saavuttaa suosiota edes lähitulevaisuudessa. Tämä ei kuitenkaan ole ollut tavoitteenakaan, joten epäonnistumisesta ei voi tässä yhteydessä puhua.

4.15 ArX

ArX[77] on Walter Landryn kehittämä GNU Archista saatuihin ideoihin perustuva hajautettu versionhallintaohjelma. Se sai alkunsa alkuperäisen Arch-projektin perustajan Tom Lordin ja silloin kehittäjänä toimineen Landryn välisistä projektin tavoitteita koskevista näkemyseroista. Koska yhteisymmärrykseen ei päästy, Landry perusti oman ArX-projektin ja kirjoitti koko silloisen koodikannan uudestaan C++-kielellä.

Landryn mukaan alkuperäinen Arch oli aivan liian monimutkainen ja vaikeasti käytettävä, joten sen käyttöliittymää oli radikaalisti yksinkertaistettava. Yksinkertaistuksen lisäksi tässä nähtiin tilaisuus tehdä myös muita parannuksia, jotka Archin sinänsä kelpoihin ominaisuuksiin yhdistettynä voisivat houkutella sovellukselle uusia käyttäjiä. Projektin tavoitteeksi asetettiin seuraavat kohdat:

- Sovelluksen on oltava helppokäyttöinen.
- Kehityshaarojen ja yhdistysmenetelmien on oltava tehokkaita.
- Sovelluksen on oltava nopea myös suurilla tietomäärillä.
- Järjestelmän on tuettava hajautettua kehitystä.
- Tietojen eheys on varmennettava kryptografisesti.
- Ohjelman on toimittava useassa eri käyttöjärjestelmässä.

Kuten tavoitteista voi nähdä, ArX pyrkii korjaamaan sekä Archin käytettävyysongelmia että perinteisiä CVS:n ajoilta periytyneitä rajoitteita, mm. tarjoamalla tuen tiedostojen ja hakemistojen uudelleennimeämiselle. Erään näkemyksen mukaan ArX on muuttunut alkuperäisestä versiostaan jo niin paljon, ettei sitä enää voi edes lukea osaksi Arch-perhettä[78].

[77] http://www.nongnu.org/arx/ (viitattu 16.8.2007).
[78] http://lists.gnu.org/archive/html/arx-users/2005-03/msg00010.html

4.15.1 Ominaisuudet

ArX on GPL-lisenssin alainen vapaa ohjelma, joka toimii yleisimmissä käyttöjärjestelmissä. Monista uuden sukupolven versionhallintaohjelmista poiketen se ei vaadi koko kehityshistorian tallentamista paikalliseen työhakemistoon. Tämän sijaan työhakemistosta voidaan viitata ulkopuolisiin tietovarastoihin, joten kehityshistoriaa voidaan säilyttää paikallista tilaa säästävästi vaikkapa erillisellä palvelimella.

Tietovarastoihin pääsee käsiksi usealla eri tavalla. Yksinkertaisimmillaan tietovarasto on paikallinen, jolloin sitä käytetään tiedostojärjestelmän kautta. Muualla sijaitseviin tietovarastoihin päästään monella eri verkkoprotokollalla, kuten HTTP:llä, SSH:lla ja FTP:llä. Kaikki muutokset tallennetaan atomisesti, joten mahdolliset virheet eivät riko tietovarastoa.

4.15.2 Toiminta

ArX tallentaa tiedostojen eri versiot käyttämällä eteneviä deltoja. Koska niiden käyttö hidastuu versioiden määrän kasvaessa, ArX mahdollistaa myös ns. *välimuistin* (engl. cache) käytön. Siinä järjestelmä hakee tietovarastosta halutun version ja tallentaa sen kokonaisena erilliseen pakettiin, jolloin kyseisen version myöhempään hakemiseen riittää tallennetun paketin purkaminen. Lopputulos on nopeampi kuin useiden deltojen perättäinen soveltaminen, mutta se myös lisää tilankulutusta. Välimuistin käyttö on peräisin alkuperäisestä Archista, ja se muistuttaa toiminnaltaan Subversionin yhteydessä esiteltävien hyppydeltojen käyttöä sillä poikkeuksella, että tallennettava perusversio on käyttäjän valittavissa.

ArX tukee kahta yhdistämistapa, joista oletuksena käytetään perinteistä 3-way-mergeä. Toinen tapa on kerätä kaikki yhdistettävät muutokset yhteen ja muodostaa niistä yksi atominen muutos.

Yhdistämistoiminnon lisäksi ArX tukee ns. *replay*-toimintoa. Sen avulla muutoksia voidaan hakea paikalliseen tietovarastoon yksi kerrallaan. Mikäli jokin muutos aiheuttaa konfliktin, se pitää korjata manuaalisesti. Tämän jälkeen toiminto jatkuu normaalisti.

Replayn avulla voidaan myös valita, mitä muutoksia halutaan ottaa mukaan. Koska ArX ei kuitenkaan pidä yllä tietoa mukaan otetuista muutoksista, nämä tiedot on annettava yhdistämis- ja replay-toiminnoille aina niitä käytettäessä. Ellei ohitettavia muutoksia mainita, kyseiset toiminnot yrittävät hakea kaikki muutokset, ohitettavat mukaanlukien.

4.15.3 BitKeeper-kriisin vaikutus ArXiin

6. huhtikuuta 2005 listalla ilmoitettiin, että BitKeeperin käyttö Linux-kehityksessä tulee todennäköisesti loppumaan. Archia pidettiin todennäköisimpänä vaihtoehtona, mutta tilanteessa nähtiin myös mahdollisuus ArXille. Kaikki eivät kuitenkaan pitäneet tilaisuutta merkittävänä, sillä ArXilla oli vain muutamia käyttäjiä[79].

15. huhtikuuta 2005 Kevin Smith aloitti listalla keskustelun GITistä. Hän ihmetteli Linusin päätöstä jättää tiedostojen uudellennimeämisen seuranta kokonaan pois, mutta totesi, että kokonaisuutena Linusin esittämät ominaisuudet näyttivät menevän pidemmälle kuin mitkään hänen aiemmin näkemänsä[80]. Lyhyessä vastauksessaan Landry totesi idean kuitenkin näyttävän epäluotettavalta[81].

4.15.4 Nykytila

ArXin nykytilan selvittämiseksi käytiin läpi ohjelman sähköpostiarkisto[82] ajanjaksolta 1.1.2005–11.7.2007.

Joulukuussa 2005 käyty mielenkiintoinen keskustelu uudesta ArX 3:sta näyttäisi jääneen viestiarkiston viimeiseksi merkittäväksi aiheeksi. Vuoden 2006 aikana listalle tuli enää 26 viestiä, ja vuoden 2007 kahden ensimmäisen neljänneksen perusteella (vain kuusi viestiä) voidaan jo päätellä, ettei mielenkiinto ArXia kohtaan ole enää kovin suurta. Uuden ArX 3:n julkaisu voisi tietenkin elvyttää mielenkiinnon, mutta sitä saadaan vielä odotella – sitä on muutettu viimeksi 23. tammikuuta 2006[83]. Nähtäväksi jää, pystyykö ArX saamaan taakseen paljon käyttäjiä, vai jääkö se uudempien kilpailijoiden jalkoihin.

4.16 Codeville

Codeville[84] on Bram ja Ross Cohenin[85] kehittämä uudesta yhdistämisalgoritmista alkunsa saanut hajautettu versionhallintaohjelma. Sen tavoitteena on olla helppokäyttöinen ja mahdollisimman skaalautuva.

[79] http://lists.gnu.org/archive/html/arx-users/2005-04/msg00013.html
[80] http://lists.gnu.org/archive/html/arx-users/2005-04/msg00032.html
[81] http://lists.gnu.org/archive/html/arx-users/2005-04/msg00033.html
[82] http://lists.gnu.org/archive/html/arx-users/
[83] http://lists.gnu.org/archive/html/arx-changes/2006-01/msg00001.html (viitattu 29.8.2007).
[84] http://codeville.org/ (viitattu 13.8.2007).
[85] Näistä veljeksistä Bram tunnetaan erityisesti BitTorrent-protokollan kehittäjänä.

Codeville on avoimen lähdekoodin ohjelma. Vuoteen 2005 asti se käytti omaa lisenssiä, mutta 5. toukokuuta 2005 julkaistusta versiosta 0.1.11 lähtien se on ollut BSD-lisenssin alainen.

4.16.1 Ominaisuudet

Alkuperästään johtuen Codevillen merkittävin ominaisuus on sen käyttämä tehokas yhdistämismenetelmä. Nykyinen ohjelmaversio tosin olettaa kaikkien tiedostojen olevan tekstimuotoisia, mutta binääritiedostojen tuen on tarkoitus valmistua ennen versiota 1.0, jonka pitäisi valmistuessaan tarjota ainakin seuraavat ominaisuudet:

- Rajoittamattomat kehityshaarat ja yhdistämiset.
- Yksinkertainen käyttöliittymä.
- Nopea myös suurten projektien yhteydessä.
- Vahva autentikointi.
- Toimivuus ilman verkkoyhteyttä.
- Tehokkaat muutostenesitys- ja historianselaustoiminnot.
- Muuttumaton ja varmennettavissa oleva historia.
- Tiedostojen ja hakemistojen uudelleennimeäminen osaa käsitellä kaikki konfliktit ja erikoistapaukset.
- Toimii yleisimmissä käyttöjärjestelmissä.

Suurin osa näistä tavoitteista on jo saavutettu, mutta versioon 1.0 on vielä jonkin verran matkaa.

4.16.2 Toiminta

Codeville tallentaa versiotiedot työhakemiston alle *.cdv*-nimiseen alihakemistoon. Komento `cdv init` luo hakemiston ja lisää sinne monia tiedostoja. Hakemiston sisältöön ei pidä koskea käsin, vaan ainoastaan Codevillen kautta.

Tietovaraston alustamisen jälkeen annetaan käyttäjätiedot. Suosituksen mukaisesti tunnuksena käytetään sähköpostiosoitetta: `cdv set user atr@localhost`.

Uudet tiedostot laitetaan versionhallintaan kahdessa osassa. Ensin tiedosto merkitään lisättäväksi komennolla `cdv add main.c`. Lopuksi varsinainen lisäys tapahtuu komennon `cdv commit` avulla. Toiminnon yhteydessä voidaan antaa tehtyä lisäystä kuvaava seloste.

Tiedostoja ei tarvitse lukita muutosten ajaksi. Tehtyjä muutoksia voidaan tarkastella komennolla `cdv diff`. Yleiskuvan muutetuista tiedostoista saa komennon `cdv status` avulla. Muutokset tallennetaan tietovarastoon komennolla `cdv commit`, joka tässäkin tapauksessa kysyy selostetta, kuten lisäyksenkin yhteydessä.

Codeville tukee tiedostojen lisäyksiä, uudelleennimeämisiä ja poistoja. Nämä tosin eroavat muista muutoksista siinä, että näitä toimintoja ei voi ainakaan toistaiseksi perua.

Kehityshaarojen ja muualla sijaitsevien tietovarastojen käyttö vaatii `cdvserver`-nimisen palvelinsovelluksen käyttöä. Palvelimen konfigurointi vaatii varsin paljon työtä, mikä poikkeaa useimmista muista tutkituista ohjelmista. Rajallisen tutustumisajan vuoksi käyttöesimerkki joudutaan rajoittamaan jo selostettuun, eli mm. kehityshaarojen ja yhdistämistoiminnon selostus on jätettävä kirjallisuudesta löytyvän tiedon varaan.

4.16.3 Alkuperäinen yhdistämisalgoritmi

Alun perin Codevillessä käytettiin perinteiseen 2-way-mergeen perustuvaa yhdistämisalgoritmia, missä tiedoston kahta eri versiota käsiteltiin rivi kerrallaan. Codevillen algoritmi poikkesi perinteisestä siinä, että se osasi käsitellä myös seuraavan esimerkin kaltaiset ristiriitatilanteet, joissa jokin muutos on tehty eri versioissa eri riville.

```
1. A|A
2. X|Y
3. B|B
4. Y|
```

Perinteinen 2-way-merge onnistuu selvittämään esimerkin rivit 1, 3 ja 4, mutta rivi 2 aiheuttaa konfliktin. Tässä vaiheessa Codeville käyttää seuraavaa algoritmia:

- Jos vasemmanpuoleista muutosta ei ole tehty oikealle puolelle, vasemmalla tehty muutos voittaa.

- Jos oikeanpuoleista muutosta ei ole tehty vasemmalle puolelle, oikealla tehty muutos voittaa.
- Jos molemman puolen muutokset voittavat, tulee konflikti.
- Jos kumpikaan muutos ei voita, tulee konflikti.

Esimerkkitilanteessa rivillä 2 olevaa vasemmanpuoleista muutosta ei ole tehty oikealle puolelle, joten muutos X voittaa. Oikealla puolella oleva muutos Y on jo otettu mukaan rivillä 4, joten se ei voita. Lopputuloksena muutos X voittaa, joten lopulliseksi riviksi tulee:

```
1. A
2. X
3. B
4. Y
```

Algoritmi toimii yleensä melko hyvin, mutta jotkut tilanteet ovat sille vaikeita. Esimerkiksi seuraava klassinen esimerkki[86] on ongelmallinen monille yhdistysalgoritmeille, kuten perinteiselle 2/3-way-mergelle ja tälle Codevillen alkuperäiselle yhdistysalgoritmille.

```
    A
   / \
  B   C
  |\ /|
  | x |
  |/ \|
  B'  C'
   \ /
    ?
```

Esimerkissä A:sta haarautuneet versiot B ja C on yhdistetty kahdella eri tavalla siten, että toiseen versioon on hyväksytty B:n muutokset ja toiseen C:n muutokset. Jos nyt näistä muodostuneet versiot B' ja C' halutaan yhdistää, lopputulos riippuu siitä, mikä versio katsotaan muutosten esi-isäksi. Jos esi-isäksi valitaan B, lopputulokseksi saadaan C'. Toisessa vaihtoehdossa lopputulokseksi saadaan B'.

[86] Esimerkkitapaus tunnetaan englanniksi mm. nimillä *criss-cross merge* ja *ambiguous clean merge*. http://revctrl.org/AmbiguousCleanMerge (viitattu 19.10.2007).

4.16.4 Precise Codeville -yhdistysmenetelmä

Edellä esitetyn kaltaisia vaikeita tilanteita varten Codevilleen on suunniteltu uutta yhdistämisalgoritmia, jonka pitäisi selvitä useimmista ongelmatilanteista. Uuden algoritmin nimeksi on varattu *Precise Codeville Merge* (lyhenne pcdv), mutta ainakaan vielä yksikään ehdokas ei ole tätä nimitystä itselleen lopullisesti ansainnut. Jäljempänä käsiteltävä *Simple Weave Merge* -algoritmi oli kuitenkin ensimmäinen, joka tätä nimitystä on väliaikaisesti käyttänyt[87].

Uudelle algoritmille on asetettu vaatimus siitä, että peräkkäisten onnistuneiden yhdistämisten lopputulosten on pysyttävä samoina riippumatta siitä, missä järjestyksessä yhdistämiset on tehty. Lisäksi sen on oltava riittävän matemaattinen, jotta sen eri ominaisuuksia pystyttäisiin todistamaan täsmällisesti. Tällä hetkellä paras ehdokas etsityksi algoritmiksi on Simple Weave Merge, joka toimii seuraavan esityksen mukaisesti.

- Jokaiselle tiedoston riville annetaan yksilöllinen tunniste. Tämä tapa on saanut vaikutteita Darcsin rivientäsmäystoiminnosta.
- Jokainen tiedostoversio sisältää tiedot poistetuista riveistä. Kun kaksi versiota yhdistetään, lopputuloksesta poistetaan kaikki ne rivit, jotka on poistettu jommasta kummasta versiosta.
- Rivit eivät saa vaihtaa järjestystä eri versioissa. Jos esimerkiksi versiossa 1 rivit ovat järjestyksessä AB, ei järjestys BA ole sallittu missään versiossa.

Viimeisen kohdan varmistamiseksi eri versioissa esiintyviltä riveiltä vaaditaan ns. täydellistä järjestystä, mitä varten jokaiselle riville lasketaan ns. järjestystunniste (engl. ordering identifier), jolla niiden oikea järjestys mahdollisessa tasatilanteessa voidaan selvittää. Järjestystunnisteena on kolmikko (d, r, n), missä d on uuden revision ja versiohistorian alun etäisyys pisintä reittiä myöten laskettuna, r on uusin revisionumero ja n on kyseisen rivin rivinumero uusimmassa revisiossa. Järjestystunnisteita verrataan arvo kerrallaan siten, että kolmikon toiseen tai kolmanteen jäseneen siirrytään vain silloin, jos edelliset vertailut eivät riitä erottamaan niitä toisistaan.

Kahden version yhdistäminen tapahtuu seuraavasti. Ensin molemmat versiot jaetaan osiin yhteisten rivien kohdalta. Jokainen osa järjestetään yllämainitun jär-

[87] http://revctrl.org/SimpleWeaveMerge (viitattu 19.10.2007).

jestystunnisteen mukaisesti, minkä jälkeen jokaisen rivin poisto tai lisäys määritetään muutaman päättelysäännön mukaisesti. Järjestyksessä otetaan huomioon myös poistetut rivit. Lopuksi tarkistetaan, ettei missään kohdassa tule konfliktia eli tilannetta, jossa kumpikin versio "voittaisi" samanaikaisesti. Käytetyt päättelysäännöt ovat seuraavat:

- Jos rivi on molemmissa versioissa, se otetaan mukaan.
- Jos rivi on poistettu jommassa kummassa versiossa, sitä ei oteta mukaan.
- Jos rivi on uusi jommassa kummassa versiossa, eikä sitä ole poistettu toisessa versiossa, se otetaan mukaan.

Seuraava esimerkki havainnollistaa tilannetta.

Alkuperäinen versio sisältää sanan $KANA$. Toinen versio muuttaa sen sanaksi $KANI$ ja kolmas version sanaksi $KONI$:

```
  K
  A
  N
  A
 / \
K   K
A   O
N   N
I   I
```

Versioiden yhdistäminen tapahtuu seuraavasti. Ensin versiot jaetaan osiin yhteisten kirjainten kohdalta. Esimerkkitapauksessa muodostuu vain yksi osa, *K*:n ja *N*:n väliin. Tämän jälkeen osion rivit järjestetään. Lopuksi kaikkiin riveihin sovelletaan edellämainittuja päättelysääntöjä:

```
K (mukana molemmissa, otetaan mukaan)
A (poistettu oikeanpuoleisessa, ei oteta mukaan)
O (uusi oikealla, otetaan mukaan)
N (mukana molemmissa, otetaan mukaan)
I (mukana molemmissa, otetaan mukaan)
```

Lopuksi tarkistetaan mahdolliset konfliktit. *K*:n ja *N*:n välisessä osassa oikeanpuoleiset muutokset voittivat molemmat kohdat, joten konfliktia ei tule. Lopullinen versio on siis seuraavanlainen:

```
K
O
N
I
```

Koska rivien on oltava järjestyksessä, ne on järkevää myös tallentaa samassa järjestyksessä, mikä johtaa weave-muodon käyttöön.

4.16.5 BitKeeper-kriisin vaikutus Codevilleen

BitKeeper-kriisi ei vaikuttanut Codevilleen suoraan. Epäsuora vaikutus näkyi GIT-versionhallintaohjelman käsittelynä.

Codevillen kehittäjät eivät nähneet GITiä merkittävänä. Bram Cohen totesikin viestissään[88], että Codevillelle on tärkeämpää hyvin tehty versionhallinta, kuin se, kuinka tunnettu nimi sitä suosittelee. Hän jatkaa viestiään kritisoiden GITin kannattajien väitteitä:

> Basically all claims to technical superiority Git has right now are based on it only doing a very limited set of things. Once it supports branching, merging, and a network protocol, [...] any advantages it has are completely gone.

Kaiken kaikkiaan BitKeeper-kriisin vaikutukset jäivät Codevillen osalta vähäisiksi, mutta uudet versionhallintaohjelmat johtivat kilpailun kovenemiseen.

4.16.6 Nykytila

Codevillen nykytilan selvittelyä varten käytiin läpi kaikki codeville-devel-listan[89] viestiarkiston viestit ajanjaksolta 13.4.2005–30.9.2007, sovelluksen www-sivusto ja sen dokumentaatiota sisältävän wiki-sivuston sisältö. Viimeksi mainittua on vandalisoitu jonkin verran, joten muokkaaminen vaatii rekisteröitymisen. Epätoivottavaa materiaalia on kuitenkin vielä nähtävissä.

[88] http://mail.off.net/pipermail/codeville-devel/2005-April/000013.html
[89] http://mail.off.net/pipermail/codeville-devel/

Uusin julkaistu versio on 0.8.0 heinäkuulta 2007[90]. Wiki-sivulla olevan TODO-listan[91] perusteella voi päätellä, että kehityksen on tarkoitus jatkua ainakin versioon 1.0, ellei pidempäänkin.

Kaikesta innovatiivisuudestaan huolimatta Codevillen kohtaloksi näyttää jäävän melko pieni suosio, mikä johtuu kilpailevien ohjelmien saavuttamasta menestyksestä. On tosin mahdollista, että Codevillen toimiviksi osoittautuneet ideat siirtyvät muihin ohjelmiin muodossa tai toisessa, mikä takaa jonkinasteisen näkyvyyden Codevillen kehittäjille.

4.17 Darcs

Darcs (David's Advanced Revision Control System) on David Roundyn kehittämä hajautettu versionhallintaohjelma, jonka perustana on ns. *päivitysteoria* (engl. Theory of Patches)[92].

Darcs sai alkunsa, kun David Roundy etsi sopivaa versionhallintaohjelmaa verkossa pelattavaa bridge-peliään varten. Alun perin hän käytti CVS:ää, mutta oli nyt etsimässä sille korvaajaa. Vaihtoehtoina olivat Subversion ja Arch[93]. Vaihtoehtojen puntaroinnin jälkeen Roundy päätyi lopulta käyttämään Archia, vaikka ei ollutkaan siihen täysin tyytyväinen. Arch-kehittäjien kanssa käydyt kehityskeskustelut nostivat kuitenkin esille uusia ideoita, jotka johtivat myöhemmin Darcsin ja pian esiteltävän ns. päivitysteorian syntyyn.

4.17.1 Ominaisuudet

Darcs on GPL-lisenssin alainen vapaa ohjelma. Sen ensimmäiset versiot toteutettiin C++-kielellä, mutta niiden debuggaaminen osoittautui turhan vaikeaksi. Nykyisissä versioissa toteutuskielenä on Haskell. [36] Ensimmäinen virallinen versio julkaistiin huhtikuussa 2003[94].

Darcsin perustana on päivitysteorian lisäksi näkemys siitä, että versionhallinta voisi olla paljon yksinkertaisempaa, kuin mitä se yleensä on. Tähän pyritään tekemällä sovelluksesta mahdollisimman älykäs ja sellainen, että se osaa kommunikoida

[90] http://mail.off.net/pipermail/codeville-devel/2007-July/000211.html (viitattu 16.10.2007).
[91] http://codeville.org/doc/ToDoList
[92] http://darcs.net/ (viitattu 1.8.2007).
[93] http://osdir.com/Article2571.phtml (viitattu 3.8.2007).
[94] http://www.haskell.org/pipermail/haskell-cafe/2003-April/004139.html

käyttäjän kanssa[95]:

- Darcs on hajautettu, joten jokaisella käyttäjällä on samat toimintamahdollisuudet. Minkäänlaista erottelua palvelimen ja asiakkaan tai muokkausoikeudellisen ja -oikeudettoman käyttäjän välillä ei tehdä.
- Darcs on interaktiivinen, joten se osaa pyytää käyttäjältä lisätietoja ja tarjota asianmukaisia vaihtoehtoja. Näin käyttäjä tietää koko ajan, mitä ollaan tekemässä.
- Darcsin älykäs muutostenkäsittely mahdollistaa toiminnot, jotka eivät muuten olisi mahdollisia.

Esitetyistä ominaisuuksista voi selvästi huomata, että Darcsista on pyritty tekemään mahdollisimman käyttäjäystävällinen. Interaktiivisuus on ideana osoittautunut niin toimivaksi ratkaisuksi, että siitä on otettu mallia myös muihin sovelluksiin, kuten GITiin. Viimeinen kohta puolestaan viittaa myöhemmin esiteltävään päivitysteoriaan.

4.17.2 Toiminta

Hakemistosta tehdään Darcs-tietovarasto komennolla `darcs init`. Uudet tiedostot lisätään kahdessa osassa. Ensin ne merkitään lisättäväksi komennon `darcs add` avulla. Tämän jälkeen lisätyt tiedostot tallennetaan tietovarastoon komennolla `darcs record`.

Mikäli tietovarasto on jo olemassa jossain muualla, sen voi ottaa käyttöön komennolla `darcs get /a/b/repo`, missä */a/b/repo* tarkoittaa tietovaraston sijaintia. Tietovarasto voi sijaita paikallisessa tiedostojärjestelmässä tai verkossa, jolloin siihen pääsee käsiksi mm. HTTP-protokollan avulla. Nykyinen Darcs ei suoranaisesti tue aliprojekteja, mutta sallii sisäkkäisten tietovarastojen käytön.

Tiedostoja voi muokata työhakemistossa suoraan, sillä niitä ei lukita. Tehtyjä muutoksia voidaan tarkastella komennon `darcs whatsnew` avulla. Valmiit muutokset tallennetaan tietovarastoon samoin kuin tiedostoja lisättäessä. Oletuksena `darcs record` toimii interaktiivisesti, kysyen käyttäjältä jokaisen muutoksen yhteydessä, otetaanko se mukaan tehtävään muutostiedostoon. Tämä mahdollistaa muutosten tallentamisen loogisissa kokonaisuuksissa.

[95] http://wiki.darcs.net/DarcsWiki (viitattu 12.10.2007).

Mikäli muualla sijaitsevaan tietovarastoon on tullut muutoksia, paikallinen tietovarasto päivitetään komennolla `darcs pull`. Paikalliset muutokset voidaan puolestaan päivittää etätietovarastoon `darcs push` -komennon avulla. Mahdolliset konfliktitilanteet ratkeavat *Darcs Merge* -yhdistysalgoritmilla[96], joka osaa mukauttaa päivityksen tuomat muutokset nykytilaan sopiviksi seuraavaksi esiteltävää päivitysteoriaa soveltamalla.

4.17.3 Päivitysteoria

Darcsin perustana oleva *päivitysteoria* (engl. Theory of Patches) on David Roundyn kehittämä teoria sellaisesta päivitysten uudelleenjärjestelystä, joka ei muuta niiden vaikutusta päivitettävään kohteeseen[97]. Teoria sai alkunsa Roundyn ja Archin kehittäjän Tom Lordin välisistä keskusteluista, ja se on herättänyt mielenkiintoa myös tieteellisissä piireissä, kun Andres Löh, Wouter Swierstra ja Daan Leijen saivat siitä alkusysäyksen yleisemmän teorian kehittämiselle. [41] Seuraavassa on lyhyt esitys päivitysteorian sisällöstä[98].

Päivitys P tekee jonkin muutoksen. P:llä on aina jokin konteksti eli tila, jossa se toimii.

P:n tilan muodostavat alkuperäinen kohde ja ne päivitykset, joita sovelletaan ennen P:tä.

Päivitykset voidaan yhdistää, mikäli ne ovat peräkkäisiä. Jos P_1:n kontekstina on alkuperäinen kohde, johon on sovellettu päivitystä P_2, voidaan P_1 ja P_2 yhdistää yhdeksi kokonaisuudeksi:

$$P_1 + P_2 \longrightarrow P_1 P_2 \tag{4.1}$$

Tämä päivitys tekee saman muutoksen, kuin alkuperäiset kaksi päivitystä, mutta sen kontekstina on alkuperäinen kohde. Yhdistetyn päivityksen toiminnan voi ymmärtää lukemalla sen oikealta vasemmalle: ensin toteutetaan P_2:n muutos ja sitten P_1:n muutos.

Päivitykset, joilla on sama konteksti, ovat rinnakkaisia. Rinnakkaisuus voidaan kuvata näin:

[96] http://revctrl.org/DarcsMerge (viitattu 6.11.2007).

[97] http://osdir.com/Article2571.phtml (viitattu 3.8.2007).

[98] Tämä osio perustuu suurimmaksi osaksi Darcsin ohjekirjan päivitysteoriasta kertovaan lukuun: http://darcs.net/manual/node8.html (viitattu 2.8.2007).

$$P_1 || P_2 \tag{4.2}$$

Päivitykset ovat kääntyviä: P:n käänteispäivityksellä P^{-1} on se ominaisuus, että yhdistetty päivitys PP^{-1} ei muuta mitään.

Yhdistetynkin päivityksen voi kääntää:

$$(P_1 P_2)^{-1} = P_2^{-1} P_1^{-1} \tag{4.3}$$

Päivitysten kontekstin voi vaihtaa kahdella tavalla. Ensimmäinen tapa on vaihtaa kahden peräkkäisen päivityksen paikkaa:

$$P_2 P_1 \longleftrightarrow P_1' P_2' \tag{4.4}$$

Tässä P_1' ja P_2' kuvaavat samoja muutoksia kuin P_1 ja P_2, mutta niiden konteksti on vaihtunut.

Toinen tapa kontekstivaihtoon on rinnakkaisten päivitysten yhdistäminen: Jos $P_1 || P_2$, niin yhdistämisen (merkitään $\Longrightarrow$) tuloksena on joko P_1' tai P_2' siten, että

$$P_2' P_1 \longleftrightarrow P_1' P_2 \tag{4.5}$$

Yhdistämisen tuloksena syntyneiden peräkkäisten päivitysten on oltava sellaisia, että niiden paikat voidaan vaihtaa.

4.17.4 BitKeeper-kriisin vaikutus Darcsiin

BitKeeper-kriisi ei vaikuttanut Darcsiin suoranaisesti, mutta sen tuloksena syntyneet ohjelmat nostattivat sen verran mielenkiintoa, että myös Darcs-kehittäjät ajautuivat mukaan keskusteluun. David Roundyn mielestä GIT oli pelkkä tietokanta, joka ei sellaisenaan soveltunut versionhallintaan. Hän oli kuitenkin vaikuttunut sen nopeudesta ja pohti, voisiko Darcs jotenkin hyötyä siitä[99].

Yksi Roundyn näkemistä GITin tuomista mahdollisuuksista oli eri versionhallintaohjelmien välinen yhteentoimivuus. Jos kaikki ohjelmat osaisivat toimia GITin kanssa, sitä voisi käyttää yhteisenä välimuotona täydellisen yhteentoimivuuden aikaansaamiseksi. Hänen mukaansa GITin vetovoima on jo siinä, että Linus Torvalds käyttää sitä[100].

[99] http://lists.osuosl.org/pipermail/darcs-devel/2005-April/001814.html
[100] http://lists.osuosl.org/pipermail/darcs-devel/2005-April/001982.html

Roundyn haaveena oli saada Linux-kehittäjät siirtymään GITin päällä toimivan Darcsin käyttäjiksi. Hän ehdotti yhteistyötä GIT-kehittäjien kanssa, tiedustellen mm. ohjelman lisenssiä ja mahdollisuutta tehdä siitä erillinen kirjasto, jota Darcs voisi hyödyntää[101]. Yhteistyö näyttäisi tuottaneen tulosta, sillä helmikuun lopulla 2005 Juliusz Chroboczek lisäsi Darcsiin tuen GIT-tietovarastojen lukemista varten[102].

4.17.5 Nykytila

Darcsin nykytilan selvittämistä varten käytiin läpi ohjelman sähköpostiarkiston[103] viestit ajanjaksolta 2.1.2005–26.9.2007. Ohjelman uusin vakaa versio on 1.0.9 kesäkuulta 2007.

Darcs on käyttökelpoinen hajautettu versionhallintaohjelma, mutta se ei ole kyennyt saavuttamaan kovin suurta suosiota. Tämä johtunee parista seikasta. Vaikka mielenkiinto Darcsia kohtaan oli alussa melko suuri[104], se ei kyennyt kilpailemaan uudempien versionhallintaohjelmien kanssa. Erään blogin sanoin: "Darcs is so 2005"[105].

Toisena esteenä voidaan pitää Darcsin toteutuskieltä. Ellei käyttäjällä ole Haskellia asennettuna tietokoneelle, kynnys Darcsin asentamiseen on suurempi, kuin esimerkiksi Pythonilla toteutetun Mercurialin kohdalla.

Vaikka Darcs vetoaakin käyttäjiinsä mm. teoreettisen mielenkiinnon ansiosta, se ei yksinään riitä nostamaan ohjelmaa uudempien kilpailijoidensa rinnalle. Tämä ei kuitenkaan tarkoita epäonnistumista, sillä Darcsin käyttämän päivitysteorian nostattama mielenkiinto on omalta osaltaan palauttanut versionhallintaohjelmille kuuluvaa asemaa tieteellisen tutkimuksen piirissä.

4.18 Monotone

Monotone [29][106] on Graydon Hoaren kehittämä hajautettu versionhallintaohjelmisto, joka tarjoaa yksinkertaisen tiedostopohjaisen transaktionaalisen versiovaraston, ja tukee verkkoriippumattomia toimintoja. Se toteuttaa kaikki tarvitsemansa

[101] http://lists.osuosl.org/pipermail/darcs-devel/2005-April/001923.html
[102] http://lists.osuosl.org/pipermail/darcs-devel/2005-April/002107.html
[103] http://lists.osuosl.org/pipermail/darcs-users/
[104] Esim. http://bc.tech.coop/blog/050710.html (viitattu 5.11.2007).
[105] http://kvardek-du.kerno.org/2007/07/all-cool-kids-are-using-git.html (viitattu 5.11.2007).
[106] http://monotone.ca/ (viitattu 13.8.2007).

toiminnot itse, joten se ei ole riippuvainen ulkoisista sovelluksista[107].

4.18.1 Ominaisuudet

Monotone on hajautettu järjestelmä, ja se käyttää omaa tehokasta peer-to-peer-synkronointiprotokollaa. Se on alusta alkaen suunniteltu mahdollisimman luotettavaksi ja turvalliseksi, mikä näkyy mm. siinä, että kryptografiaa käytetään asiakaspään RSA-sertifikaattien lisäksi myös versioiden nimeämiseen.

Monotone käyttää kehityshaarojen yhdistämisessä 3-way-mergeä kahdessa vaiheessa. Ensimmäisessä vaiheessa toimitaan muutosjoukon tasolla, eli tunnistetaan hakemistorakenteeseen tulleet muutokset, kuten hakemistojen ja tiedostojen lisäykset ja poistot, sekä uudelleennimeämiset. Tämän jälkeen siirrytään yksittäisten tiedostojen tasolle, jolloin muutoksia käsitellään rivi kerrallaan Codevillen ideoihin perustuvan *Mark Merge* -algoritmin[108] mukaisesti. Mikäli jomman kumman vaiheen aikana esiintyy konflikteja, ongelman ratkaisu siirretään käyttäjän tehtäväksi.

Muutosten levittämiseen käytetään *netsync*-nimistä verkkoprotokollaa. Erillisiä palvelimia ei tarvita, sillä jokainen asiakas voi toimia myös palvelimena. Alun perin Monotone tuki myös muita protokollia, kuten HTTP:tä ja SMTP:tä, mutta myöhemmin niistä on luovuttu erinäisten ongelmien vuoksi.

Monotone toimii useissa eri käyttöympäristöissä. Se on lisensoitu GPL:n alaisuuteen, eli se on vapaa ohjelma.

4.18.2 Toiminta

Monotonessa käyttäjä luo kehitysprojekteja varten itselleen sekä uuden tietokannan että salasanasuojatun tunnuksen. Tietokannan luonti onnistuu komennolla `mtn db init --db=~/Proj.mtn`, ja tunnuksen esimerkiksi komennolla `mtn genkey atr@localhost`. Olemassa olevat tunnukset näkee komennon `mtn list keys` avulla.

Uusi projekti asetetaan käyttökuntoon komennolla `mtn --db=~/Proj.mtn --branch=main setup proj`, joka luo uuden työhakemiston (tässä *proj*) ja sen sisään alihakemiston *_MTN*, jonne ohjelma tallentaa omia tietojaan. Tämän jälkeen voidaan siirtyä työhakemistoon ja aloittaa projekti.

[107] Monotone tosin käyttää ulkoista Boost-kirjastoa, mutta senkin voi linkittää sovellukseen staattisesti, jolloin ulkoista riippuvuutta ei synny.

[108] http://revctrl.org/MarkMerge (viitattu 8.11.2007).

Uusien tiedostojen lisäys tapahtuu kahdessa osassa. Ensin tiedostot merkitään lisättäväksi komennon `mtn add main.c` avulla, minkä jälkeen varsinainen lisäys tehdään komennolla `mtn commit`. Commit-toiminnolle voidaan haluttaessa antaa lokiteksti, missä kerrotaan tehdyistä muutoksista[109]. Lopuksi ohjelma kysyy salasanan.

Tehtyjä muutoksia voidaan tarkastella komennolla `mtn diff`, ja aiempien muutosten lokitekstit näkee komennolla `mtn log`. Komennolla `mtn status` voidaan puolestaan selvittää työhakemiston nykytila, eli katsoa, onko tiedostoja lisätty, poistettu tai muokattu. Komento näyttää myös käytössä olevan kehityshaaran.

Monotone tukee erillisten kehityshaarojen käyttöä. Olemassa olevat haarat näkee komennolla `mtn ls branches`, ja uuden haaran aloitus tapahtuu yksinkertaisesti antamalla commit-toiminnolle uuden haaran nimi: `mtn commit --branch=haara`. Komennon jälkeen työhakemisto käyttää uutta haaraa, joten seuraavien muutosten tallennusten yhteydessä sitä ei tarvitse enää mainita. Siirtyminen kehityshaarojen välillä tapahtuu komennolla `mtn update`, ja niissä tehtyjä muutoksia voi yhdistellä komennolla `mtn merge`.

Muutosten levittämiseen eri kehittäjien välillä käytetään luotettavaa verkkoyhteyttä, mitä varten kehittäjien on vaihdettava keskenään tunnustensa julkisia avaimia. Monotone tarjoaa tähän tarkoitukseen omat komentonsa. Lopuksi julkistettavassa tietovarastossa käynnistetään erillinen palvelin, joka huolehtii verkkotoiminnoista. Paikallinen työhakemisto päivitetään komennolla `mtn sync`, minkä jälkeen kaikki muutokset ovat paikallisessa tietokannassa, eikä verkkoyhteyttä enää tarvita ennen seuraavaa synkronointia.

4.18.3 Merkitys

BitKeeper-kriisin yhteydessä Linus Torvalds mainitsi Monotonen mahdollisena korvaavana järjestelmänä. Vaikka sitä ei lopulta valittukaan, sen vaikutus Linusin myöhemmin kehittämään GITiin on melkoisen suuri. Esimerkiksi GITin tapa käsitellä tiedostoja niiden SHA-1-arvon perusteella on selvästi peräisin Monotonesta.

[109] Lokitekstin antaminen ei testattaessa toiminut aivan oikein. Monotone käyttää viestin kirjoittamiseen EDITOR-ympäristömuuttujassa annettua editoria, mutta toimi virheellisesti tilanteessa, jossa arvona oli “emacsclient -a nano”. Muissa testatuissa ohjelmissa käynnistyi joko emacs tai nano, mutta Monotonen tapauksessa tuli virhe “mtn: misuse: edit of log message failed”.

4.18.4 Nykytila

Monotonen nykytilan selvittelyä varten käytiin läpi monotone-devel-listan viestiarkiston[110] viestit ajanjaksolta 1.1.2005–30.9.2007. Ajanjakson jälkeiset tapahtumat selvitettiin ohjelman kotisivulta.

Uusin julkaistu versio on 0.37 lokakuulta 2007. Muutamien kuukausien välein tehdyt julkaisut antavat ohjelman tulevaisuudesta positiivisen kuvan. Se, miten ohjelma tulee pärjäämään kilpailijoidensa seurassa, jää nähtäväksi. BitKeeper-kriisin kärjistyttyä Linus Torvalds mainitsi Monotonen yhtenä vaihtoehtona, joten on mielenkiintoista seurata, osoittautuuko ohjelma sitä kohtaan osoitetun luottamuksen arvoiseksi.

4.19 Superversion

Superversion[111] on Stefan Reichin kehittämä Change Set -pohjainen usean käyttäjän hajautettu versionhallintajärjestelmä. Se on toteutettu Java-kielellä, ja sopii graafisen käyttöliittymänsä ansiosta myös sellaisille käyttäjille, joilla ei ole aiempaa kokemusta versionhallintaohjelmista.

Käytetyn ohjelmointikielen ansiosta Superversion toimii useissa eri käyttöjärjestelmissä, ja on toinen kahdesta tarkasteluun päätyneestä täysin graafisesta sovelluksesta. WWW-sivujensa perusteella sen on tarkoitus olla kilpailukykyinen kaupallisten versionhallintaohjelmien kanssa.

4.19.1 Ominaisuudet

Superversion on graafinen versionhallintaohjelma, joka mahdollistaa monien versionhallinnan perustoimintojen käytön yksinkertaisen käyttöliittymän avulla. Ohjelma käyttää transaktiopohjaista tietokantaa ja tallentaa kaikki muutokset deltoina, binääritiedostot mukaanlukien.

Graafisuutta on hyödynnetty kaikissa toiminnoissa. Versiohistorian tarkastelussa voi käyttää tekstihakua ja verrata eri versioita. Myös koko projektin kehityshistorian tarkastelu on mahdollista.

Superversion tukee avainsanojen laajennusta. Tiedostotyyppi (binääri/teksti) tunnistetaan automaattisesti. Ohjelma tukee kehityshaaroja ja niiden yhdistämistä. Eri

[110] http://lists.gnu.org/archive/html/monotone-devel/
[111] http://www.superversion.org/ (viitattu 14.8.2007).

versioille voidaan antaa kuvaavia nimiä, ja kokonaisia versioita voidaan paketoida zip-tiedostoon.

Superversion on lisensoitu GPL:n alaisuuteen.

4.19.2 Toiminta

Superversion käsittelee kokonaisia projekteja. Uusi projekti luodaan valikosta löytyvän toiminnon avulla. Ohjelma luo projektille uuden hakemiston, jonne versiotiedot tallennetaan. Tämän lisäksi ohjelmalle kerrotaan, missä versioitavat lähdekoodit sijaitsevat.

Tiedostot lisätään lähdekoodihakemistoon tavalliseen tapaan. Superversion tunnistaa hakemistoon tehdyt muutokset ja näyttää ne käyttäjälle. Tiedostot voidaan tallentaa tietovarastoon selostetekstin kera.

Muutosten tekokin tapahtuu perinteiseen tapaan. Superversion seuraa hakemiston tilaa, joten tehtyjä muutoksia voi tarkastella milloin tahansa. Muutosten tallentaminen tapahtuu samalla tavalla kuin uusien tiedostojen lisääminen.

Ohjelma sallii työhakemiston päivittämisen mihin tahansa olemassa olevaan versioon. Saman toiminnon avulla muutokset voidaan tallentaa erilliseen kehityshaaraan. Myös yhdistystoiminto on graafinen. Versioille voidaan antaa myös kuvaavia nimiä.

Muutosten jakeluun Superversion ei tarjoa erityisiä työkaluja, mutta se tukee kokonaisten projektien levitystä. Tätä tarkoitusta varten ohjelma osaa pakata halutun version zip-pakettiin.

4.19.3 Nykytila

Superversionin nykytilan selvittelyä varten käytiin läpi superversion-discussion-listan sähköpostiarkiston[112] viestit ajanjaksolta 12.1.2005–30.9.2007. Viestien perusteella voitiin selvästi päätellä, että Superversionin aktiivinen kehitys oli lakannut, sillä viimeisin varsinainen uutinen on elokuulta 2005.

Sähköpostiarkistosta on myös havaittavissa viestien määrän romahtaminen. Esimerkiksi viimeisen vuoden aikana (elokuu 2006–elokuu 2007) viestejä on kertynyt vain 7 kappaletta, joista helmikuun postissa kysytään, onko projekti kuollut. Viestiin ei ole tullut vastausta.

Superversionin tulevaisuudesta olisi pelkkien sähköpostiviestien perusteella voi-

[112] https://lists.sourceforge.net/lists/listinfo/superversion-discussion (viitattu 14.8.2007).

nut tehdä sen johtopäätöksen, ettei sitä enää kehitetä. Tutkielman viimeistelyn yhteydessä kävi kuitenkin selväksi, että ainakin avoimen lähdekoodin projekteissa kaikenlaiseen odottamattomaan kannattaa varautua. Superversionin tapauksessa kuolleelta vaikuttava projekti saattaa vielä hyvinkin elpyä lähitulevaisuudessa, kuten sen kehittäjän viesti 10.10.2007 osoittaa[113]:

> True, development has stalled for about two years now. However, Beta 8 is still in use by some people (including me), so it seems like a waste to leave the software in beta state forever.
>
> [...]
>
> Here's a fairly realistic plan. I'll continue working on my current project, Project Prophecy, until about year's end. Afterwards, it may be Superversion's term. Can't promise anything yet though. Sorry!

Tässä tapauksessa vain aika näyttää, mikä Superversionin kohtaloksi tulee.

4.20 SVK

SVK[114] on Chia-liang Kaon Subversionin päälle kehittämä hajautettu versionhallintajärjestelmä. Sen kehitys alkoi vuonna 2003 ja vuonna 2006 se siirtyi Best Practical Solutionsin omistukseen.

SVK:n tarkoituksena on jatkaa siitä, mihin Subversion jäi. Kuten aiemmin esitettiin, Subversionin tavoitteena on ollut tarjota CVS:lle vaihtoehto, joka on mahdollisimman paljon esikuvansa kaltainen, mutta korjaa sen suurimmat puutteet. Tämän tavoitteen se on enimmäkseen saavuttanutkin, mutta lähtökohtiensa vuoksi siihen on edelleenkin jäänyt sellaisia rajoitteita, joiden vuoksi monet sovelluskehittäjät ovat siirtyneet käyttämään jotain muuta versionhallintaohjelmaa.

SVK:n kehittäjien mielestä CVS:n ja sen myötä Subversionin suurin ongelma on rajoittuminen keskitetyn tietovaraston käyttöön. Heidän mukaansa Subversion kyllä tavoittaa entiset CVS-käyttäjät, mutta muiden järjestelmien käyttäjät olisi myös otettava huomioon. Tämän vuoksi SVK:n päätehtävänä on lisätä Subversionin päälle niitä ominaisuuksia, joita nykyaikana on alettu vaatia, kuten mahdollisuutta hajautettuun kehitykseen ja parempien yhdistysmenetelmien käyttöä.

[113] http://www.superversion.org/ (viitattu 22.10.2007).

[114] http://svk.elixus.org/ (viitattu 13.8.2007),
http://rt.openfoundry.org/Foundry/Project/Forum/?Queue=82 (viitattu 13.8.2007),
http://bestpractical.com/svk/ (viitattu 13.8.2007).

4.20.1 Ominaisuudet

Koska SVK ei ole itsenäinen sovellus vaan Subversionin laajennus, se vaatii toimiakseen Subversionin kirjastojen asennuksen. Sillä ei myöskään ole omaa tietovarastoformaattia, mutta se käyttää olemassa olevia Subversionin tietovarastoja omalla tavallaan niin, että ne toimivat hajautetussa kehityksessä. SVK:ta käytettäessä tietovarastoja voidaan esimerkiksi peilata (engl. mirror), eli kopioida kokonaisina siten, että koko kehityshistoria saadaan siirtymään paikasta toiseen. Tämä mahdollistaa joidenkin Subversionissa verkkoyhteyttä vaativien toimintojen hoitamisen paikallisesti ilman verkon käyttöä.

Tietovarastoon viittaavasta lyhyestä nimestä käytetään SVK:n yhteydessä nimitystä *depot*, jolloin esimerkiksi "" (tyhjä merkkijono) tarkoittaa paikallista tietovarastoa "∼/.svk/local". Taustastaan johtuen SVK:n tietovarastoja voi lukea suoraan myös Subversionilla, ja kirjoittaminenkin onnistuu, mutta sitä kehotetaan välttämään. Tietoja tallennettaessa SVK kerää esimerkiksi yhdistämiseen liittyviä tietoja talteen, jolloin suora kirjoitus tietovarastoon SVK:n ohi voi aiheuttaa myöhemmin tilanteen, jossa automaattinen yhdistäminen ei näiden tietojen puutteen vuoksi onnistu.

SVK:n käyttämä *smerge*-yhdistysmenetelmä on saanut vaikutteita Tom Lordin alun perin Arch-järjestelmään kehittämästä star-mergestä. Se kerää ja pitää tallessa tietoja tehdyistä yhdistämisistä ja osaa käyttää näitä tietoja myöhempien yhdistämisten helpottamiseen.

SVK toimii hyvin myös muiden versionhallintajärjestelmien kanssa, kuten myös graafisten yhdistystyökalujen kanssa. Ohjelma on toteutettu Perl-kielellä ja lisensoitu sen kanssa samalla tavalla. Käytössä on kaksoislisenssi, eli samanaikaisesti Artistic-lisenssi ja GPL.

4.20.2 Toiminta

SVK tukee samoja perustoimintoja, kuin Subversion. Tietovarastossa oleva koodi saadaan ulos komennolla `svk checkout`. Komento luo uuden työhakemiston ja täyttää sen lähdekooditiedostoilla.

Tietovaraston ja työhakemiston päivittämiseen käytetään eri komentoja. Paikallinen tietovarasto päivitetään komennolla `svk sync`. Tämän jälkeen työhakemisto päivitetään komennolla `svk update`. Saman komennon avulla voidaan työhakemistoon ottaa mikä tahansa tietovarastossa oleva versio.

Uudet tiedostot lisätään versionhallintaan komennolla `svk add`. Tiedostojen muokkaus tapahtuu työhakemistossa. Tehtyjä muutoksia voidaan tarkastella komennolla `svk diff`. Valmiit muutokset tallennetaan tietovarastoon komennolla `svk commit`.

SVK tukee tiedostojen poistoja, uudelleennimeämisiä ja kopiointeja. Kopiointi ja uudelleennimeäminen osaavat säilyttää tiedostojen versio historian.

SVK osaa yhdistää tehtyjä muutoksia sekä työhakemistossa että suoraan tietovarastossa. Yhdistäminen tapahtuu komennolla `svk merge`. Tämän lisäksi SVK osaa käyttää GNU Archin kehittäjän Tom Lordin kehittämästä star-mergestä vaikutteita saanutta smergeä (smart merge), joka mm. säilyttää tietoja tehdyistä yhdistyksistä.

4.20.3 Nykytila

Nykytilan selvittämiseksi käytiin läpi ohjelman sähköpostiarkistojen[115] viestit ajanjaksolta 4.1.2005–4.9.2007.

Ohjelman uusin version on 2.0.2 heinäkuulta 2007.

SVK:n käyttämä Perl-kieli ei ole kaikkein tutuimpia suurelle osalle mahdollisista käyttäjistä, mutta sen löytyminen lähes jokaisesta Unix-järjestelmästä on niin varmaa, ettei sen käyttö pienennä mahdollisen käyttäjäkunnan määrää yhtä paljoa, kuin esim. Meta-CVS:n ja Darcsin tapauksissa.

Ohjelma näyttää sijoittuvan ketjun RCS-CVS-Subversion jatkoksi. Tässä suhteessa se olisi looginen seuraava vaihe siirryttäessä pois CVS:n käytöstä hajautettujen versionhallintaohjelmien pariin. Subversion on jo ottanut paikkansa CVS:n seuraajana, joten on mielenkiintoista seurata, mihin suuntaan seuraava askel otetaan.

4.21 Pastwatch

Pastwatch [7, 73][116] on Benjie Chenin väitöskirjassaan [7] vuonna 2004 esittelemä versionhallintajärjestelmä, jonka tarkoituksena on mahdollistaa tiedostojen käyttö laajalle levittäytyneissä ympäristöissä[117].

[115] Alkuperäinen lista oli osoitteessa http://lists.openfoundry.org/wws/arc/svk-dev. 24.7.2006 lista siirtyi osoitteeseen http://lists.bestpractical.com/pipermail/svk-devel.

[116] http://pdos.csail.mit.edu/pastwatch/ (viitattu 1.8.2007).

[117] [“. . .] to support wide-area read/write file sharing.”

4.21.1 Ominaisuudet

Pastwatchin perusominaisuudet vastaavat CVS:n ominaisuuksia, mutta niiden lisäksi ohjelma tarjoaa joitakin kehittyneempiä toimintoja:

- Tietovarasto on hajautettu.
- Muutosten tekoon ei tarvita verkkoyhteyttä.
- Tiedostojen päivitys on nopeaa, sillä verkkoyhteyttä käytetään ainoastaan muutosten hakuun. Kaikki muut toiminnot ovat paikallisia.
- Atomiset muutokset mahdollistavat usean muutoksen käsittelyn yhtenä kokonaisuutena.

Pastwatchin mukana tulevan *cvs2past*-ohjelman avulla CVS-tietovarasto voidaan muttaa Pastwatchin käyttämäksi tietovarastoksi.

Ominaisuuksiensa perusteella Pastwatch on tyypillinen hajautettu versionhallintaohjelma. Se on lisensoitu GPL:n alaisuuteen, mikä ei tosin selviä ohjelman kotisivulta vaan sen lähdekoodipaketissa olevasta lisenssitekstistä.

4.21.2 Toiminta

Pastwatch tallentaa kehityshistorian puumaiseen rakenteeseen (branch tree), missä jokainen solmu vastaa yhtä versiota, eli koko projektin tilaa tietyllä hetkellä. Jokainen versio sisältää sekä kopiot kaikista projektin tiedostoista että viitteen edelliseen versioon. Muutosten tekoa on rajoitettu siten, että niitä voidaan tehdä vain uusimman version päälle, mutta erillisen kehityshaaran voi aloittaa mistä solmusta tahansa.

Normaalisti lineaarinen kehityshistoria muuttuu puumaiseksi joko erillisiä kehityshaaroja käytettäessä tai silloin, kun johonkin versioon tehdään muutoksia ainakin kahdessa eri tietovarastossa, jotka synkronoidaan keskenään. Tällöin historiapuuhun muodostuu kohtia, joissa jollakin solmulla on kaksi tai useampia lapsia, ja jotka käyttäjä normaalitapauksessa yhdistää takaisin yhdeksi solmuksi. Koska muutokset ovat pysyviä eikä niitä voi jälkikäteen muuttaa, “ylimääräiset” solmut merkitään yleensä yhdistämisen jälkeen epäaktiivisiksi (engl. inactive).

Tietovarastojen synkronointi eli niissä tehtyjen muutosten levitys ja yhdistäminen on Pastwatchissa asynkronista, eli se ei välttämättä tapahdu heti. Käytännössä

tämä tarkoittaa sitä, että kun tietovaraston sisältävä tietokone kytketään verkkoon, sen sisältö päivittyy ajan myötä uusimpaan versioon. Synkronoinnin lopputuloksena on tilanne, jossa jokainen tietovarasto sisältää samanlaisen historiapuun: "[... it does provide *eventual consistency*; if all disconnected users re-connect and users do not commit new changes, then eventually all users' branch trees become identical." [7]

Pastwatch tallentaa versiot Berkeley DB -tietokantaan tilannekuvina, jotka tilansäästön vuoksi pilkotaan useaan lohkoon. Jokaiselle lohkolle lasketaan sen sisällön perusteella yksilöllinen 160-bittinen tunniste, joten useassa tilannekuvassa olevat samansisältöiset lohkot tallennetaan vain kerran. Kuhunkin versioon liittyvien lohkojen tiedot löytyvät versioihin liittyvästä metadatasta.

Lohkojen tunnisteiden oikeellisuus tarkistetaan aina niiden käytön yhteydessä, joten tietovaraston rikkoutuminen tai jälkikäteen tehdyt asiattomat muutokset havaitaan ennen kuin ne aiheuttavat ongelmia.

4.21.3 Nykytila

Pastwatchin viimeisin julkaistu versio on 0.8.2 syyskuulta 2005. Ohjelman CVS:ssä (!) oleva lähdekoodi ei näytä edistyneen lainkaan maaliskuun 2006 jälkeen.

Nykytilasta ja tulevaisuudensuunnitelmista ei näytä löytyvän kovin paljon tietoa, käyttäjistä puhumattakaan. Ainoa johtolanka löytyy ohjelman kotisivulta[118], jossa mainitaan Pastwatch-projektin ylläpitämän Aqua-palvelun sulkemisesta mm. mielenkiinnon puutteen vuoksi. Tämä ei kuitenkaan estä ohjelman käyttöä, minkä vuoksi käyttäjämäärää on mahdotonta arvioida.

Pastwatchin tutkimuslähtöisestä taustasta johtuen sen tulevaisuudesta on mahdotonta sanoa mitään, mutta ainakaan lähiaikoina se ei muodosta minkäänlaista uhkaa olemassa oleville versionhallintaohjelmille.

[118] http://pdos.csail.mit.edu/pastwatch/ (viitattu 5.11.2007).

4.22 Bazaar(-NG)

Bazaar[119] on Python-kielellä kirjoitettu hajautettu versionhallintaohjelma. Sen kehitystä sponsoroivat mm. Canonical Limited[120], Ubuntun[121] perustajat ja Launchpad[122], joten sillä on suuren kehittäjämäärän lisäksi myös vahva taloudellinen tuki.

Bazaarin suunnitteluperiaatteet ovat oikeellisuus, nopeus, yksinkertaisuus ja helppokäyttöisyys, joista viimeisimpään pyritään mm. tekemällä ohjelmasta CVS:n ja Subversionin käyttäjille mahdollisimman tutun oloinen. Näiden lisäksi projektin tavoitteena on ollut saada aikaiseksi sellainen versionhallintajärjestelmä, josta myös vapaiden ohjelmien liikkeen kannattajat voisivat pitää. Projektiin osallistuvien näkemyksen mukaan hyvin tehty hajautettu versionhallintaohjelma auttaa vapaiden ohjelmien projekteja saamaan lisää kehittäjiä.

4.22.1 Uusi ja vanha, bzr ja baz

Nimellä Bazaar voidaan tarkoittaa kahta eri sovellusta. Alun perin sillä tarkoitettiin tiettyä Arch-pohjaista versionhallintaohjelmaa, mutta nykyään se viittaa myös uudempaan, alusta alkaen uudestaan suunniteltuun sovellukseen. Yhteistä näille on vain nimi. Vanhan Bazaar 1.x -sarjan sovellukset tunnetaan lyhenteellä *baz*, ja uudemman 2.x-sarjan lyhenteellä *bzr*. Uudempaa on nimitetty myös Bazaar-NG:ksi (Next Generation), mutta nykyisin sekaannuksen vaaraa ei juurikaan ole, joten sille riittää pelkkä nimitys Bazaar, jota myös tässä tutkielmassa käytetään.

4.22.2 Ominaisuudet

Käytetyn Python-kielen ansiosta Bazaar toimii useassa eri käyttöympäristössä. Virallisesti tuettuja ovat mm. Linux- ja Microsoft Windows -järjestelmät, mutta mitään teknisiä esteitä muissa järjestelmissä toimimiselle ei ole. Muiden ympäristöjen toimivuutta ei voida kuitenkaan taata, koska sitä ei ole testattu.

Bazaarin kehittäjien tavoitteena on ollut tehdä järjestelmästä sellainen, ettei se vaadi käyttäjää muuttamaan toimintatapojaan, eikä se rajoita versioitavalle datalle tehtäviä operaatioita millään tavalla. Esimerkiksi tiedostojen uudelleennimeämi-

[119] http://bazaar-vcs.org/ (viitattu 13.8.2007). Myös Bazaar-NG:n osoite http://bazaar-ng.org/ johtaa samalle sivulle.

[120] http://canonical.com

[121] http://ubuntu.com

[122] https://launchpad.net

nen onnistuu käyttöympäristön omilla perustoiminnoilla, joten käyttäjän ei tarvitse omaksua sen vuoksi uusia komentoja.

Bazaarille on asetettu seuraavat tavoitteet:

- Riittävä nopeus suurimmalle osalle projekteja.
- Tietojen on oltava turvassa.
- Ystävällisyys.
- Vapaus. (GPLv2+)

Bazaarin kehityshaarojen julkistaminen on helppoa, esimerkiksi HTTP- tai FTP-protokollaa käyttämällä. Näiden lisäksi voidaan käyttää erillistä HTTP- tai SSH-protokollaa käyttävää *smart server*iä, joka vähentää tarvittavaa verkkoliikennettä.

4.22.3 Toiminta

Bazaarin alkuaikoina versioiden tallentamiseen käytettiin perinteistä SCCS:stä peräisin olevaa weave-muotoa, joka kuitenkin osoittautui liian tehottomaksi ja virhealttiiksi ratkaisuksi[123]. Nykyisin käytössä on weaveen perustuva ns. *append-only weave* eli *knit*-muoto, missä jokaista tiedostoa kohden on oma datatiedosto, jonka loppuun uudet versiot lisätään. Menetelmä nopeuttaa versioiden tallennusta sekä vähentää datatiedostojen pirstoutumista, mutta sen toteutus on perinteistä tapaa monimutkaisempi.

Bazaarin käyttämä *Knit Merge* -yhdistämisalgoritmi[124] osaa minimoida konfliktien määrän. Eri kehityshaaroissa tehdyt identtiset muutokset eivät aiheuta ongelmia, sillä järjestelmä osaa yhdistää ne automaattisesti. Algoritmin toiminta on samantapainen kuin Codevillen yhteydessä esiteltävän Simple Weave Mergen, josta se on saanut innoituksensa.

Kehityksen yhteydessä käytetään kattavaa testausta. Kehityshaarojen oikeellisuus voidaan tarkistaa milloin vain. Tämän lisäksi eri versiot voidaan allekirjoittaa kryptografisesti, minkä jälkeen muutoksia ei voi tehdä ilman, että ne havaitaan.

Bazaarin käyttö on helppoa. Peruskäyttöön riittää vain muutama komento, eikä harvinaistenkaan komentojen kanssa tule ongelmia, koska jokainen komento on kattavasti dokumentoitu. Tarvittaessa apu löytyy sähköpostilistalta tai wiki-muotoiselta ohjesivustolta.

[123] http://bazaar-vcs.org/BzrWeaveFormat (viitattu 31.10.2007).
[124] http://bazaar-vcs.org/KnitMerge (viitattu 31.10.2007).

4.22.4 BitKeeper-kriisin vaikutus Bazaariin

Tieto BitKeeperin käytön lopetuksesta Linux-kehityksessä tuli myös Bazaar-listalle. Vastaanotto oli melko rauhallinen, eikä suurempaa keskustelua syntynyt. Tämä saattoi johtua siitä, että varsinaista keskustelua käytiin monien uutispalvelujen foorumeilla, joten mitään uutta sanottavaa ei ollut[125].

Muutama päivä GITin julkistamisen jälkeen listalla on keskustelua siitä, miten Bazaar voisi hyödyntää GITiä. Martin Pool toteaa, että on eri mieltä joistakin Linusin toteutusratkaisuista, kuten tiivistearvon laskemisesta pakatusta objektista, mutta aikoo silti käyttää joitain optimointi-ideoita[126].

4.22.5 Nykytila

Bazaarin nykytilan selvittelyä varten käytiin läpi sen sähköpostiarkiston[127] viestit ajanjaksolta 21.3.2005–30.9.2007. Uusin julkaistu versio on 0.92 marraskuulta 2007[128]. Melko säännöllinen julkaisutahti viittaa siihen, että ohjelman kehitys jatkuu aktiivisesti. Versio 1.0 näyttäisi olevan tulossa lähiaikoina.

Bazaarin tulevaisuus näyttää valoisalta. Canonicalin tarjoama taloudellinen tuki voi auttaa saavuttamaan yritysten luottamuksen, ja kehityksen avoimuus vetoaa moneen avoimen lähdekoodin suosijaan. Potentiaalisia kehittäjiä ajatellen Pythonin käyttö kehityskielenä voi tarjota jonkin verran etua esimerkiksi C-kieliseen GITiin nähden.

4.23 Bky

Bky[129] on Angel Ortegan kehittämä minimalistinen hajautettu versionhallintaohjelma. Se on toteutettu yksinkertaisena komentotulkkiskriptinä (engl. shell script, vrt. DOS-järjestelmien bat-tiedostot), ja käyttää versioiden tallentamiseen rsync-ohjelmaa. Toteutustapansa vuoksi eri versiot tallentuvat kokonaisina hakemistopuina, mutta sovellus pyrkii minimoimaan niiden viemän tilan tallentamalla muuttumattomat tiedostot kovina linkkeinä (engl. hard links).

Toiminnassaan Bky hyödyntää valmiita Unix-järjestelmistä löytyviä apuohjel-

[125] https://lists.ubuntu.com/archives/bazaar/2005q2/000072.html
[126] https://lists.ubuntu.com/archives/bazaar/2005q2/000185.html
[127] https://lists.ubuntu.com/archives/bazaar/
[128] http://bazaar-vcs.org/ (viitattu 13.11.2007).
[129] http://www.triptico.com/software/bky.html (viitattu 1.8.2007).

mia, kuten diffiä ja patchia. Se ei tarjoa osaavalle käyttäjälle varsinaisesti mitään uutta, mutta aloittelijalle se voi toimia yksinkertaisena käyttöliittymänä vaikeammin hallittaviin sovelluksiin.

Bky:n kehitys näyttää alkaneen huhtikuussa 2005. Tähän antaa viitteitä se, että järjestelmää pystyi käyttämään itsensä kehitykseen toukokuun 1. päivänä[130]. Kyseinen ajanjakso sijoittuu juuri BitKeeper-kriisin huipun jälkeiseen aikaan, mikä johtaa kysymykseen, oliko kyseisellä tapahtumalla mitään merkitystä Bky:n aloitukseen.

4.23.1 Ominaisuudet

Bky:n tärkeimpiä ominaisuuksia ovat:

Hajautus Tietovarastona toimii työhakemistossa oleva alihakemisto. Kehittäjät jakavat tekemänsä muutokset patch-tiedostojen avulla. Käytetty tapa vastaa modernien versionhallintaohjelmien, kuten Darcsin ja GITin, käyttämää tapaa.

Kevyet kehityshaarat Kehityshaaran luominen tapahtuu yksinkertaisesti kopioimalla työhakemisto uuteen hakemistoon.

Historian typistys Vanhat versiot voidaan poistaa tai siirtää muualle arkistoitavaksi järjestelmän perustyökalujen avulla.

Turvallisuus Revisiot tallennetaan tietovarastoon kokonaisina hakemistoina, joten niitä voidaan käsitellään käyttöjärjestelmän perustyökalujen avulla.

Tietovaraston muokattavuus Koska jokainen revisio tallennetaan tavallisina tiedostoina, niitä ja metadataa voidaan muokata suoraan järjestelmän perustyökaluilla.

Ei tarvetta erikoisohjelmille Kun tiedosto luodaan työhakemistoon, se kuuluu heti versionhallinnan alaisuuteen. Ohjelma osaa automaattisesti jättää tavallisimmat luodut tiedostot, kuten objektitiedostot, huomiotta. Myös .cvsignore-tiedoston käyttö on mahdollista.

Vähäiset riippuvuudet Bky on shelliskripti. Se tarvitsee perinteisten Unix-työkalujen lisäksi vain rsync- ja diff-ohjelmat. Näiden lisäksi tiedostojärjestelmän on tuettava kovia ja symbolisia linkkejä.

[130] http://www.triptico.com/download/Changelog_bky

Vaikka Bky ei tarjoa mitään erityisen innovatiivisia toimintoja, se vaikuttaa riittävältä pienimuotoiseen versionhallintaan. Sen suurimpana puutteena voidaan pitää atomisten muutosten teon puuttumista.

4.23.2 Toiminta

Hakemisto laitetaan Bky:n hallintaan käyttämällä komentoa `bky init`. Komento luo uuden `.bky`-nimisen alihakemiston, jonne versiot tallennetaan. Tämän jälkeen kaikki hakemistossa olevat tiedostot ovat automaattisesti versionhallinnan alaisuudessa[131].

Työhakemiston tilanne tallennetaan versioksi komennolla `bky commit`. Komennolle voidaan antaa myös muutosta selostava viesti. Uusi versio muodostetaan kopioimalla työhakemisto tietovarastona käytettyyn alihakemistoon *rsync*-ohjelman avulla.

Työhakemistoon tehtyjä muutoksia voidaan tarkastella joko yleisellä tasolla tai muutoskohtaisesti. Yleisellä tasolla järjestelmä näyttää, mitä tiedostoja on lisätty, poistettu tai muutettu. Muutoskohtaisesti tarkasteltuna järjestelmä näyttää tehdyt muutokset diff-muodossa. Muutoskohtainen tarkastelu onnistuu myös eri versioiden välillä. Käytetty muutosformaatti ei tue binääritiedostojen muutosten esittämistä.

Mikä tahansa versio voidaan ottaa ulos tietovarastosta. Tällöin työhakemistossa tehdyt muutokset häviävät. Järjestelmä tukee myös tageja, joilla versioille voidaan antaa kuvaavia nimiä. Nimien pitää olla sellaisia, että ne kelpaavat käytetyn tiedostojärjestelmän tiedostonimiksi.

Bky mahdollistaa koko työhakemiston synkronoinnin eri tietokoneiden välillä. Käytännössä tämä tapahtuu *rsync*-komennon avulla, mihin järjestelmä tarjoaa yksinkertaisen push/pull-käyttöliittymän.

Bky:n mukana tulevan *cvs2bky*-skriptin avulla voidaan siirtää tietoja CVS-tietovarastosta Bky:hyn. Skripti vaatii toimiakseen *cvsps*- ja *patch*-ohjelmat.

[131] Poikkeuksen muodostavat ne tiedostot, jotka komento *rsync -C* katsoo parhaaksi ohittaa. Käytännössä ohitettavat tiedostot ovat samat, mitkä CVS:kin ohittaisi.

4.23.3 Nykytila

Bky oli alun perin lisensoitu GPL:n alaisuuteen, mutta sen lisenssi vaihtui 25. kesäkuuta 2006 BSD-lisenssiksi[132]. Nykyinen versio (lokakuussa 2007) ohjelmasta on 13. elokuuta 2007 julkaistu 1.1.0. Bky:llä ei vaikuttaisi olevan sähköpostilistaa, joten sen kehittämisestä vastannee vain yksi henkilö.

Yksinkertaisuudestaan huolimatta Bky muistuttaa melko paljon moderneja hajautettuja versionhallintaohjelmia. Vaikka toteutustapa skriptejä käyttäen on hieman arveluttava, perusideat paikallisine toistensa kanssa synkronoitavine tietovarastoineen vastaavat nykyajan käsitystä hajautetusta versionhallinnasta.

Vaikka Bky:n näyttääkin jäävän vaille suurempaa menestystä, se on silti mielenkiintoinen lisä muiden versionhallintajärjestelmien joukkoon. Sillä on kuitenkin kyseenalainen kunnia olla ainoa tarkasteltu sovellus, joka onnistui testeissä hävittämään tietoja, joten sen käyttö nykymuodossa on varsin riskialtista.

4.24 FastCST

FastCST (Fast Change Set Tool)[133] on Zed A. Shaw'n kehittämä kokeellinen versionhallintajärjestelmä, jonka päätavoitteita ovat turvallisuus, hajautus ja nopeus. Sen mottona on lause "Distribute like an anarchist, but accept like a fascist".

Ensimmäinen FastCST:n versio toteutettiin C-kielellä, mutta myöhemmin kieli vaihtui Rubyksi. Tavoitteena on tehdä kieliriippumaton järjestelmä, joten käytetyt tiedostomuodot ja verkkoprotokollat pyritään dokumentoimaan hyvin.

4.24.1 Ominaisuudet

FastCST on muutosjoukkopohjainen (engl. Change Set based). Siinä revisiolla tarkoitetaan valmiin muutosjoukon nimeä, joten käsitteitä muutosjoukko ja revisio käytetään melkein synonyymeinä. Uuden muutosjoukon teko aloitetaan erillisellä komennolla, jonka jälkeen kaikki ennen lopetuskomentoa tehdyt muutokset ryhmittyvät saman revision osaksi.

FastCST mahdollistaa ns. *lisäosien* (engl. plugin) teon. Lisäosat jaetaan kahteen luokkaan, *komentoihin* ja *triggereihin*.

Komennot toimivat samoin kuin muut perustoiminnot. Ne ottavat parametreja

[132] http://www.triptico.com/download/Changelog_bky (viitattu 25.9.2007).
[133] http://www.zedshaw.com/projects/fastcst/index.html (viitattu 3.9.2007).

ja pääsevät käsiksi järjestelmän ohjelmointirajapintaan. Triggereiden avulla puolestaan voidaan suorittaa toimintoja ennen komentoja ja komentojen jälkeen. Ne siis tavallaan ympäröivät komentoja.

FastCST käyttää kehittäjänsä keksimää uutta delta-algoritmia, joka on tekijänsä mukaan sekä nopea että pieniä deltoja tuottava.

FastCST:n omana lisenssinä on GPL, mutta se sisältää myös muiden lisenssien alaista koodia[134]:

> The suffix array code backing it is licensed under the Plan9 license [...] Also the software directory has a lot of software that is used.

Sovelluksen tarttuva GPL-lisenssi ei koske lisäosia, mikäli ne on toteutettu itse, eivätkä sisällä muualta kopioituja osia.

4.24.2 Toiminta

FastCST tallentaa tietonsa lähdekoodihakemiston sisälle luotuun `.fastcst`-alihakemistoon. Komento `fcst` ilman parametreja käynnistää komentotilan, missä versiohistoria esitetään tiedostojärjestelmän tapaisena puurakenteena. Puussa liikkuminen tapahtuu komentojen `apply` ja `undo` avulla. Näistä ensimmäinen siirtyy parametrina annettuun versioon, ja jälkimmäinen kumoaa viimeisimmän `apply`:n. Komennot ovat analogisia tiedostojärjestelmissä käytettyjen komentojen `cd dir` ja `cd ..` suhteen, ja myös `ls`-komennolle löytyy oma vastineensa, `list`.

Muutosten teko ilmoitetaan eksplisiittisesti `begin`-komennon avulla. Komento luo uuden revision ja antaa sille yksilöllisen tunnisteen, minkä jälkeen käyttäjä voi tehdä haluamansa muutokset tai yhdistämiset. Lopuksi annetaan `finish`-komento, joka kiinnittää revision lopulliseen muotoonsa. Kaikki näiden komentojen välillä tehdyt toiminnot tulevat osaksi lopullista revisiota.

FastCST:n yhdistämistoiminto ei ole vielä valmis. Tällä hetkellä eri revisioita voidaan yhdistää vain silloin, kun niissä tehdyt muutokset ovat riittävän erillisiä, eivätkä aiheuta konflikteja. Mikäli muutokset aiheuttavat ristiriitoja, niitä ei voida yhdistää ollenkaan[135].

[134] http://www.zedshaw.com/zedshaw.com/projects/fastcst/faq.html (viitattu 25.8.2007).

[135] FastCST 0.6.5:n lähdekoodin README-tiedostossa sanotaan näin: "Merging is implemented, but conflict resolution is not yet. It currently will not let you resolve conflicts and refuses to do the merge."

Muutosten jakelu onnistuu sähköpostin lisäksi myös järjestelmän sisäänrakennetun web-palvelimen avulla. Palvelimen käyttö kuitenkin estää tietovaraston samanaikaisen käytön muuhun toimintaan, joten sitä ei voi suositella jatkuvaan käyttöön. Kolmas vaihtoehto on muutostiedostojen jakelu FTP-palvelinten välityksellä, mihin FastCST tarjoaa oman `get`-komennon.

4.24.3 Non-Linear Suffix Tree Deltas

FastCST käyttää kehittäjänsä Zed A. Shaw'n kehittämää NSTD-algoritmiä[136] (Non-Linear Suffix Tree Deltas) muutosten laskemiseen. Shaw toteaa algoritmin olevan sekä nopean että tehokkaan. Algoritmi käyttää apunaan *suffiksipuuta* (engl. suffix tree), joka on tapa tallentaa merkkijono siten, että sen päätemerkkijonojen[137] etsiminen on nopeaa. Algoritmin toiminta on seuraava:

1. Alkuperäisestä tiedostosta muodostetaan suffiksipuu.
2. Muokatun tiedoston alusta etsitään pisin täsmäävä merkkijono. Merkitään tulos *täsmänneeksi* (engl. matched).
3. Jatketaan etsintää etsimällä pisin ei-täsmäävä merkkijono. Merkitään tulos *lisättäväksi* (engl. inserted).
4. Jatketaan kohdasta 2, kunnes koko tiedosto on käsitelty.

Algoritmin tuloksena on lista täsmänneistä ja lisätyistä merkkijonoista. Seuraavassa esimerkissä algoritmia sovelletaan merkkijonojen $KOIRA$ ja $KISSA$ muutosten esittämiseen:

1. Muodostetaan sanan $KOIRA$ suffiksipuu. Puusta löytyvät merkkijonot $KOIRA$, $OIRA$, IRA, RA ja A.
2. Aloitetaan sanan $KISSA$ käsittely. Kirjain K täsmää suffiksin $KOIRA$ alkuun, joten merkitään *täsmäys K*. Täsmättäväksi merkitään pisin mahdollinen osa, eli jos koiran tilalla olisi ollut vaikkapa kirahvi, täsmääväksi osaksi olisi merkitty KI.

[136] http://www.zedshaw.com/projects/nstd/index.html

[137] Esimerkiksi sanan "banaani" päätemerkkijonot ovat "banaani", "anaani", "naani", "aani", "ani" "ni" ja "i".

3. Jatketaan kirjaimesta I etsimällä pisin ei-täsmäävä merkkijono. Koska kirjain täsmää merkkijonon IRA alkuun, se aiheuttaisi tyhjän *lisäyksen*, jota ei kuitenkaan merkitä. Tässä tapauksessa I ei "kulu", joten samaa kirjainta käsitellään myös seuraavassa vaiheessa.

4. Kirjain I täsmää suffiksin IRA ensimmäiseen kirjaimeen, joten merkitään *täsmäys* I.

5. Jatketaan kirjaimesta S. Se ei täsmää mihinkään suffiksiin, joten se merkitään muistiin ja siirrytään seuraavaan kirjaimeen, joka on myös S. Sekään ei täsmää, joten muistiin tulee merkkijono SS. Kirjain A kuitenkin täsmää, joten muistissa oleva merkkijono SS merkitään *lisättäväksi*, mutta A ei kulu pois.

6. Lopuksi viimeinen kirjain A täsmää, joten merkitään *täsmäys* A.

Algoritmin lopputuloksena syntyvä delta esittää muutokset säilytettävien merkkien ja lisättävien merkkien avulla:

```
= K
= I
+ SS
= A
```

Kuten esimerkistä voidaan havaita, merkkijonoja ei yritetä täsmätä suffikseihin vaan ainoastaan niiden kirjaimiin.

Algoritmi toimii kehittäjänsä ilmoittamalla tavalla, mutta sillä on omat haittapuolensa. Suffiksipuun luonti isoille tiedostoille kuluttaa paljon muistia, eikä algoritmin avulla tehtyä deltaa voida käyttää tiedostojen yhdistämiseen.

4.24.4 Nykytila

FastCST ei ole vielä (lokakuussa 2007) valmis. Esimerkiksi yhdistämistuki on konfliktinratkaisun osalta niin puutteellinen, ettei sitä voi pitää edes käyttökelpoisena. Nykyiseen lähdekoodiin on kuitenkin merkitty lukuisia kehitysideoita, joiden toteuttamisen jälkeen tilanne voisi olla aivan toinen.

Viimeisin julkaistu versio FastCST:stä on 0.6.5 maaliskuulta 2005. Koska ohjelman Ruby-kielinen versio aloitettiin saman vuoden helmikuussa, ja sen mainittiin

sisältävän enemmän toimintoja kuin alkuperäinen versio[138], voidaan todeta, että uuden julkaisun tulo näyttää vähintäänkin epävarmalta. Mikäli sellainen jossain vaiheessa ilmestyy, sen ominaisuuksista ei voi vielä sanoa mitään varmaa.

4.25 GIT

GIT[139] on Linus Torvaldsin alun perin omaan käyttöönsä suunnittelema ja toteuttama versionhallintaohjelmisto, joka on saanut vaikutteita mm. Monotonesta ja kaupallisesta BitKeeperistä. Sen tavoitteita ovat nopeus, tehokkuus ja toimivuus isoissa projekteissa[140].

GIT on taustansa ja lähtökohtiensa vuoksi erityisen mielenkiintoinen versionhallintaohjelma. Sen kehitys alkoi Linusin ym. tutustuttua suureen määrään olemassa olevia versionhallintaohjelmia, ja päätyessä siihen, ettei mikään niistä vastannut odotuksia kaikilta osin. Tämän vuoksi GITin kehityksessä on otettu alusta alkaen huomioon aiempien sovellusten puutteet ja rajoitukset, jolloin ne on osattu kiertää tai niiden esiintyminen on saatu minimoitua. Tämä ei kuitenkaan tarkoita sitä, että GIT olisi täydellinen versionhallintaohjelma; voidaan kuitenkin sanoa, että se kuuluu ehdottomasti alansa parhaimmistoon.

4.25.1 Ominaisuudet

GITin tavoitteena on ollut alusta asti tarjota sopiva järjestelmä Linux-ytimen kehitystarpeisiin, mikä näkyy sen tarjoamissa ominaisuuksissa varsin hyvin:

Vahva tuki ei-lineaariselle kehitykselle GIT tarjoaa loistavan tuen useiden kehityshaarojen käyttöön ja niiden yhdistämiselle. GITin mukana tulevat tehokkaat työkalut mahdollistavat helpon navigoinnin ei-lineaarisessa kehityshistorissa ja kehityshistorian visualisoinnin.

Hajautettu kehitys GIT antaa BitKeeperin ja SVK:n mukaisesti jokaiselle kehittäjälle oman paikallisen tietovaraston koko kehityshistoriasta. Tehdyt muutokset kopioidaan tietovarastosta toiseen. Muualta tulleet muutokset liittyvät osaksi

[138] http://www.zedshaw.com/projects/fastcst/faq.html (viitattu 23.10.2007).

[139] Linus Torvalds: "git" can mean anything, depending on your mood: [...] "Global information tracker": you're in a good mood, and it actually works for you. [...]

[140] http://git.or.cz/ (viitattu 15.1.2007).

paikallista tietovarastoa erillisinä haaroina, jotka voidaan yhdistää pääkehityshaaraan, kuten tavalliset paikalliset haaratkin. Tietovarastoihin pääsee käsiksi tehokkaalla GIT-protokollalla, joka toimii myös ssh-yhteyden avulla. Perinteinen HTTP-protokolla on myös käytössä, joten tietovaraston julkaisua varten ei tarvitse asentaa ja konfiguroida erillisiä palvelimia.

Isojen projektien tehokas käsittely GIT on erittäin nopea ja sopii myös isoille projekteille, joissa on pitkä historia. Tavallisissa toiminnoissa GIT on yleensä jopa kertaluokkaa nopeampi kuin mikään muu versionhallintaohjelma, joissakin tapauksissa jopa useita kertaluokkia nopeampi. GIT käyttämä versiohistorian tallennustapa on myös täysin omaa luokkaansa verrattuna muihin avoimen lähdekoodin versionhallintaohjelmiin.

Kryptografisesti autentikoitu versiohistoria GIT tallentaa versiohistorian siten, että minkä tahansa yksittäisen version[141] nimi riippuu paitsi versiosta itsestään, myös koko sitä edeltävästä historiasta. Kun versio on julkaistu, sitä ei voi enää muuttaa ilman, että muutos havaitaan. Versioiden lisäksi myös tagit voidaan allekirjoittaa kryptografisesti.

Unix-mainen suunnittelu GIT seuraa Unix-traditiota siten, että sen toteutus koostuu useasta pienestä työkaluohjelmasta, jotka on kirjoitettu C-kielellä [1]. Tämän lisäksi mukana on joukko skriptejä, jotka tarjoavat miellyttävän käyttöliittymän alemman tason työkaluihin. Tällaisen suunnittelutavan mukaisesti eri komponentteja voidaan yhdistellä helposti ja näin toteuttaa uusia toimintoja.

GITin kehityshistoriaa on erityisen helppo seurata, koska se on tallennettu lähes alusta alkaen GITiin itseensä. Projekti alkoi Linusin muistelun mukaan 3.4.2005, ja ensimmäinen säilynyt muutos on tehty 7.4.2005[142]. Kaikki tämän jälkeen kertynyt historia on tallessa ja julkisesti saatavilla.

4.25.2 BitKeeper-kriisin vaikutus GITiin

BitKeeper-kriisin vaikutus GITiin on valtava, sillä ilman kyseistä tapahtumaa koko ohjelmaa ei olisi edes aloitettu. GIT olikin alun perin nopeasti tehty hätäratkaisu, jo-

[141] GIT käyttää versiosta nimeä “commit”.

[142] http://marc.info/?l=git&m=117254154130732 (viitattu 1.10.2007).

ka osoittautui kuitenkin niin hyväksi, ettei siitä kannattanut enää siirtyä mihinkään muuhun versionhallintaohjelmaan.

On tärkeää muistaa, että edellämainittu kriisi oli kytenyt pinnan alla jo kauan ennen lopullista esiintuloaan. Linus oli tehnyt taustalla paljon työtä pyrkiäkseen sovittelemaan eri näkemyksiä, mutta ei onnistunut siinä täysin. Hän tiesi jo varhaisessa vaiheessa, mitä olisi edessä, joten hänellä oli hieman aikaa tutustua erilaisiin vaihtoehtoihin. Mikään vaihtoehto ei kuitenkaan ollut riittävä, joten hän päätyi uuden ohjelman luomiseen.

4.25.3 Kehityksen alkuvaiheet

GIT-kehityksen alkuvaiheen viesteistä voi selvästi huomata, että projekti on tiukasti Linus Torvaldsin kontrolloima. Viestissään 15.4.2005 Linus kuitenkin esittää perustelunsa tälle tavanomaista tiukemmalla käytökselle[143]:

> I'm really not that much of an SCM guy. I detest pretty much all SCM's out there, and while it's been interesting to do 'git', I've done it because I was forced to, and because I really wanted to put _my_ needs and opinions first in an SCM, and see how that works. That's why I've been so adamant about having "philosophy", because otherwise I'd probably just end up with yet another SCM that I'd despise.

Alkuaikojen tiukka kontrolli selvästikin höltyy myöhemmin, kun tärkeimmät kehittäjät ovat osoittaneet sisäistäneensä Linusin peräänkuuluttaman filosofian.

GITin nykyinen ylläpitäjä Junio C Hamano otti osaa kehitykseen alusta alkaen. Hänen ensimmäinen viestinsä kehityslistalle on päivätty 12.4.2005, mutta hänen nimensä esiintyy GITin kehityshistoriassa jo päivää aiemmin. Muut aikaisessa vaiheessa mukana olleet kehittäjät olivatkin lähinnä tuttuja nimiä Linux-ympyröistä, joka näkyi mm. keskusteluiden sisällöstä: ext3-tiedostojärjestelmän indeksointi, DMA-kontrollerien nopeudet, little-endian-/big-endian-arkkitehtuurit ym. Listaa seuraamalla keskustelijoiden Linux-tausta tulee esille jopa niin voimakkaasti, että huhujen, joiden mukaan GIT on varta vasten räätälöity vain Linux-kehitykseen, alkuperä voidaan helposti tunnistaa[144].

[143] http://www.gelato.unsw.edu.au/archives/git/0504/0623.html (viitattu 1.10.2007).

[144] Aivan alussa GIT todellakin oli tarkoitettu vain Linux-kehitykseen, mutta myöhemmin tilanne on muuttunut. Huhut ovat kuitenkin sitkeitä, joten niitä esiintyy vieläkin mm. eri keskustelupalstoilla.

4.25.4 Erikoiset ratkaisut

13.4.2005 Linus puolusteli tekemäänsä epätavallista ratkaisua[145]:

> So I do agree that it's strange to do the SHA1 on the compressed object. But that was very much by design, and I think I have a really good reason for it.
>
> [...]
>
> I dunno. I feel pretty strongly that I made a good choice, but on the other hand, I've felt that way about things I ended up being totally wrong about, so don't let that stop you from arguing. Maybe my reasons aren't as good as I thought they were, so feel free to shoot holes in my parade ;)

Viikon kuluttua Linus huomasikin olleensa väärässä, ja myönsi virheensä 20.4.2005 lähettämässään viestissä[146]:

> I converted my git archives (kernel and git itself) to do the SHA1 hash _before_ the compression phase.
>
> So I'll just have to publically admit that everybody who complained about that particular design decision was right. Oh, well.

Tämän seurauksena GITiin tehtiin muutos, joka rikkoi sen yhteensopivuuden kaikkiin edellisiin versioihin, mutta jonka kaikki, Arch-kehittäjä Tom Lordia lukuunottamatta, katsoivat oikeaksi ratkaisuksi.

GITissä käytetty Monotonesta peräisin oleva tapa nimetä muutokset niiden SHA-1-arvolla joutui myös itsessään kritiikin kohteeksi, kun kehityslistalle 16.4.2005 lähetetyssä viestissä ilmaistiin huoli käytetyn algoritmin turvallisuudesta. Viesti johti laajaan keskusteluun kryptografisista funktioista ja niiden heikkouksista, mutta johtopäätökseksi tuli, ettei huoleen ollut ainakaan tässä tapauksessa aihetta. Linus muistutti, että turvallisuus syntyy aivan muusta kuin hyvin valituista algoritmeista.

Yksi erikoisena pidetty ratkaisu oli tallentaa tiedostojen uudet versiot pakattuina mutta kokonaisina. Perinteisesti muutokset oli totuttu tallentamaan tilaa säästävinä deltoina, mutta GITin tapauksessa jokainen yhdenkin merkin muutos vaati koko tiedoston tallentamisen. Erityisesti Mercurialin kehittäjä Matt Mackall kritisoi tätä ratkaisua ja piti sitä kestämättömänä[147]. Linus kuitenkin tyrmäsi kritiikin to-

[145] http://www.gelato.unsw.edu.au/archives/git/0504/0212.html (viitattu 1.10.2007).
[146] http://www.gelato.unsw.edu.au/archives/git/0504/1339.html (viitattu 1.10.2007).
[147] http://www.gelato.unsw.edu.au/archives/git/0504/2502.html (viitattu 3.10.2007).

teamalla, ettei tilansäästö kuulunut hänen GITille asettamiensa kolmen tavoitteen joukkoon.

Linusin kielteisestä asenteesta huolimatta mm. Chris Mason ja Nicolas Pitre jatkoivat erilaisten tilansäästömenetelmien kokeilua. Näiden tuloksena valmistui xdelta-algoritmiin perustuva pakkausmenetelmä, joka tilankäytön minimoinnin lisäksi myös nopeutti GITin toimintaa entisestään, jolloin myös Linus kiinnostui asiasta. Linusin ja GITin tulevan ylläpitäjän Junio C Hamanon tiiviin yhteistyön tuloksena GITistä tulikin lopulta nopeimman lisäksi myös kaikkein vähiten tilaa käyttävä avoimen lähdekoodin versionhallintaohjelma[148]:

> [...] the last area of the system we worked together before he handed the project over to me, was what we call packfile. This is the file format and data structure designed for efficient data storage and network transfer. A packfile stores the files compressed — one copy of a file is stored compressed and other files with similar contents are expressed as difference to that copy. This format is so efficient that it turns out that git has the smallest disk footprint to represent the same history among modern SCM systems.

Uuden menetelmän ansiosta Linux-ytimen BitKeeperistä GITiin siirretyn kehityshistorian koko putosikin ennätyksellisesti 2,5 gigatavusta 227 megatavuun[149]. Menetelmän tehokkuus selittyy yksinkertaisesti sillä, ettei siinä rajoituta vain eteneviin tai takeneviin deltoihin, vaan deltat lasketaan niin, että niistä saadaan paras mahdollinen tilansäästö. Operaatio on raskas, joten se suositellaan tehtävän ajastetusti järjestelmän muiden huoltotoimenpiteiden yhteydessä.

4.25.5 Toiminta

Varsinkin GITin alkuaikoina sen käyttö oli varsin monimutkaista, mutta nykyään käyttäjäystävällisyyteen on kiinnitetty enemmän huomiota. Joidenkin edistyksellisten toimintojen käyttö vaatii vieläkin jonkin verran opiskelua (tai yritystä ja erehdystä), mutta perustoiminnot eivät enää vaadi omien skriptien tekoa, niin kuin ennen.

Sekä aloitettavat että olemassa olevat projektit saadaan GITin alle komennolla `git init`, joka luo työhakemiston sisälle *.git*-nimisen tietovaraston. Tietovarasto

[148] http://gitster.livejournal.com/10002.html (viitattu 20.10.2007).

[149] http://www.gelato.unsw.edu.au/archives/git/0505/3947.html (viitattu 20.10.2007).

sisältää koko projektin historian sekä muuta tarvittavaa metadataa, kuten projektiin liittyvät asetukset ja eri toimintojen yhteydessä suoritettavat skriptit. Mikäli käyttöön halutaan saada muualla oleva projekti, se onnistuu esimerkiksi komennolla `git clone git://www.yritys.fi/projekti.git`. GITin oman protokollan lisäksi tiedonsiirto onnistuu mm. http:n, https:n, ssh:n ja rsyncin avulla.

Työhakemistossa olevia tiedostoja voidaan muokata tavalliseen tapaan järjestelmän perustyökaluilla, mutta muista järjestelmistä poiketen niitä ei voi tallentaa suoraan tietovarastoon. GITissä tehdyt muutokset lisätään ensin `git add`-komennolla *indeksiin*, jonka sisältö tallennetaan myöhemmin atomisesti komennolla `git commit`. Käytetty tapa mahdollistaa vertailut sekä työhakemiston ja indeksin (`git diff`), indeksin ja viimeisimmän muutoksen (`git diff –cached`) että työhakemiston ja viimeisimmän muutoksen välillä (`git diff HEAD`).

Tiedostojen kopiointeja ja nimien vaihdoksia ei seurata erikseen, koska ne voidaan päätellä myöhemmin. Monissa tapauksissa päättely onnistuu myös silloin, kun tiedostoa on nimeämisen jälkeen muokattu:

```
  diff --git a/kissa.txt b/koira.txt
similarity index 86%
rename from kissa.txt
rename to koira.txt
```

Tietojen siirto eri tietovarastojen välillä onnistuu komennoilla `git push` ja `git pull`, joista jälkimmäinen on oikeastaan yhdistelmä `git fetch` + `git merge`. Yhdessä tietovarastossa voi olla useita eri kehityshaaroja, joiden välillä siirrytään komennolla `git checkout`.

Normaalista poiketen GITin tietovarastoa on aika ajoin huollettava, ettei sen toiminta hidastuisi. Aiemmin käyttäjän piti muistaa ajaa komennot `git repack` ja `git prune` silloin tällöin pakatakseen tietovaraston sisältämät objektit tilaa säästävään muotoon ja poistaakseen ylimääräiset objektit. Nykyisin käyttäjän ei kuitenkaan tarvitse huolehtia tästä, sillä nämä ja muut huoltotoimenpiteet tekevä `git gc` ajetaan automaattisesti monien operaatioiden päätteeksi.

4.25.6 Yhteistyö muiden järjestelmien kanssa

Kehityksen alkuaikoina suurin osa listalla käydyistä keskusteluista käsitteli järjestelmän tehokkuutta ja tilankäyttöä, joita vertailtiin mm. BitKeeperin, CVS:n ja Sub-

versionin vastaaviin ominaisuuksiin. Nopeuden suhteen GIT osoittautui ylivoimaiseksi, mutta tilankäytössä se hävisi niille selvästi.

14.4.2005 tuleva Mercurial-kehittäjä Matt Mackall ilmoitti listalla kehittämästään tehokkaasta tallennusmenetelmästä, jolla tilankulutus ja nopeus saatiin jopa BitKeeperin vastaavia ominaisuuksia paremmiksi[150]. Viikkoa myöhemmin hän ilmoitti kehittämästään versionhallintaohjelman prototyypistä, Mercurialista, joka aikaisesta kehitysvaiheestaan huolimatta vaikuttaa lupaavalta. 26.4.2005 Mackall lähetti listalle viestin, jossa vertailtiin Mercurial 0.3:ta ja GITiä[151]. Testitulokset herättivät jonkin verran kiinnostusta, mutta keskustelun sisältö vaihtui pian käsittelemään ext3-tiedostojärjestelmän soveltuvuutta erilaisille datamäärille.

Huhtikuun puolessa välissä 2005 alkoi keskustelu GITin ja Darcsin välisestä yhteistyöstä. Darcsin pääkehittäjä David Roundy oli aluksi innokas käyttämään GITiä järjestelmänsä osana[152], mutta joutui myöhemmin toteamaan, että ohjelmien toimintafilosofiat ovat niin erilaisia, että yhteistyössä jouduttaisiin tekemään kompromisseja[153]:

> The trouble is that the philosophy of darcs and git are about as orthogonal as one can come. Git treats the content as fundamental, where in darcs the changes are fundamental. [...] when interacting with git, we either need to restrict darcs to only describe changes in a way that can be uniquely determined by a parent and child, or we need to have extra metadata somewhere.

Vaikeuksista huolimatta 26.4.2005 julkistettiin "Git-aware darcs"[154], joka kuitenkin osoittautui epäkäytännölliseksi: "To put it mildly, Darcs is not optimised for that sort of usage."[155] Vaikeuksista näytetään kuitenkin selvinneen, joten nykyään sen tulevaisuus näyttää varsin lupaavalta[156].

20.4.2005 Archin kehittäjä Tom Lord lähetti listalle viestin otsikolla "[ANNOUNCEMENT] /Arch/ embraces 'git'" , jossa hän kertoi aikeistaan hyödyntää GITiä omassa järjestelmässään. Lordin innostuksesta huolimatta Codeville-kehittäjä Bram Co-

[150]http://www.gelato.unsw.edu.au/archives/git/0504/0379.html (viitattu 1.10.2007).
[151]http://www.gelato.unsw.edu.au/archives/git/0504/2062.html (viitattu 2.10.2007).
[152]http://www.gelato.unsw.edu.au/archives/git/0504/0681.html (viitattu 1.10.2007).
[153]http://www.gelato.unsw.edu.au/archives/git/0504/1189.html (viitattu 2.10.2007).
[154]http://www.gelato.unsw.edu.au/archives/git/0504/1945.html (viitattu 2.10.2007).
[155]http://www.gelato.unsw.edu.au/archives/git/0504/2250.html (viitattu 3.10.2007).
[156]Katso esim. http://wiki.darcs.net/DarcsWiki/DarcsGit, viitattu 6.11.2007.

hen oli kuitenkin skeptinen GITin suhteen[157]:

> I'm trying to tell you that the amount of time between now and when a system as nice as BitKeeper is in use by the kernel can be dramatically reduced by either using an existing system verbatim or basing new efforts on one.
>
> If you thing that git as it exists right now is at all comparable to Monotone or Codeville you're completely delusional.

Lordin ja Cohenin esittämien äärimmäisten näkemysten väliin mahtuu paljon ideoita ja ehdotuksia, jotka ovat omalta osaltaan vaikuttaneet GITin kehitykseen.

4.25.7 Yhdistysmenetelmät

14.4.2005 Junio C Hamano lähetti listalle viestin, jonka mukana oli Perlillä toteutettu 3-way-merge GITille. Keskustelun yhteydessä Linus muistutti kaikkia seuraavasta yhdistämisperiaatteesta, jota ehdotettu menetelmä selvästi noudatti[158]:

> So the rule should be: only merge when it's "obviously the right thing". If it's not obvious, the merge should _not_ try to guess what the right thing is. It's much better to fail loudly.

Myöhemmin listalla käytiin keskustelua eri yhdistysmenetelmien ongelmatilanteista, joista yhtä on käsitelty aiemmin kohdassa 4.16.3 sivulla 116. Keskusteluun otti osaa mm. Codevillen kehittäjä Bram Cohen. GITissä edellä esitettyyn ongelmaan saatiin ratkaisu, kun Fredrik Kuivinen esitteli uuden innovatiivisen yhdistämismenetelmänsä *rekursiivisen 3-way-mergen*[159]. Siinä ongelmalliset esi-isän valinnat vältetään muodostamalla täsmälleen yksi virtuaalinen esi-isä, joka saadaan aikaan yhdistämälle kaikki mahdolliset esi-isät rekursiivisesti niin pitkälle, kunnes jäljelle jää vain yksi vaihtoehto.

Uuden yhdistämismenetelmän lisäys oli helppoa, sillä GITissä on mahdollista käyttää useita eri menetelmiä, joita sovelletaan halutussa järjestyksessä. Yleensä ensin kokeillaan nopeinta vaihtoehtoa, ja jos se ei toimi, voidaan kokeilla jotain muuta. Periaatteessa tämä mahdollistaa myös erilaisten binääritiedostojen yhdistämisen, koska jokaista tiedostotyyppiä varten voidaan tehdä oma yhdistysalgoritmi.

[157] http://www.gelato.unsw.edu.au/archives/git/0504/2185.html (viitattu 2.10.2007).
[158] http://www.gelato.unsw.edu.au/archives/git/0504/0109.html (viitattu 1.10.2007).
[159] http://www.gelato.unsw.edu.au/archives/git/0508/8209.html (viitattu 17.10.2007).

4.25.8 Kehityshistorian siistiminen

GITissä kehityshistoriaa ei voi jälkikäteen muuttaa, mutta sen voi kirjoittaa kokonaan uusiksi halutulla tavalla. Tätä varten on olemassa omat komentonsa, kuten *rebase*, jolla aiemmin tehdyt muutokset voidaan siirtää alkamaan halutusta kohdasta, ja *filter-branch*, jolla historiaa voidan muokata erittäin monipuolisesti. Esimerkiksi seuraava komento poistaa kehityshistoriasta kaikki Matti Meikäläisen tekemät muutokset[160]:

```
git filter-branch --commit-filter '
    if [ "$GIT_AUTHOR_NAME" = "Matti Meikäläinen" ];
    then
            skip_commit "$@";
    else
            git commit-tree "$@";
    fi' HEAD
```

Tällainen historian suodatus antaa osaavalle käyttäjälle runsaasti mahdollisuuksia, mutta myös vaatii GITin toiminnan varsin hyvää tuntemista. Tavalliselle käyttäjälle parempi vaihtoehto onkin interaktiivinen *rebase*-toiminto (rebase -i), missä muutokset esitetään tekstieditorissa erillisinä riveinä, joita käyttäjä voi siirrellä, yhdistellä ja poistella haluamallaan tavalla.

Seuraava käytännön esimerkki osoittaa, minkälaisissa tilanteissa tällaista kehityshistorian siistimistä oikeasti käytetään. Esitetty tapahtumasarja sijoittuu joulukuuhun 2006, ja sen tuloksena Linux-ytimeen otettiin mukaan seuraavanlainen muutos:

```
From: Jordan Crouse <jordan.crouse@amd.com>
Date: Fri, 8 Dec 2006 10:40:21 +0000 (-0800)
Subject: [PATCH] video: Get the default mode from the right database

[PATCH] video: Get the default mode from the right database
```

[160] Tarkemmin sanottuna muutoksia "sitovat" objektit (engl. commit) poistuvat, jolloin historiaan ei jää Matti Meikäläisen tekemiä muutoksia. Oletuksena näiden muutosten sisältö (engl. changes) kuitenkin säilyy, sillä ne yhdistetään niitä seuraavan historian osaksi. Tarvittaessa myös nämä sisällöt voi poistaa, mutta silloin käyttäjän on osattava "paikata" niiden historiaan jättämistä aukoista aiheutuvat ongelmat.

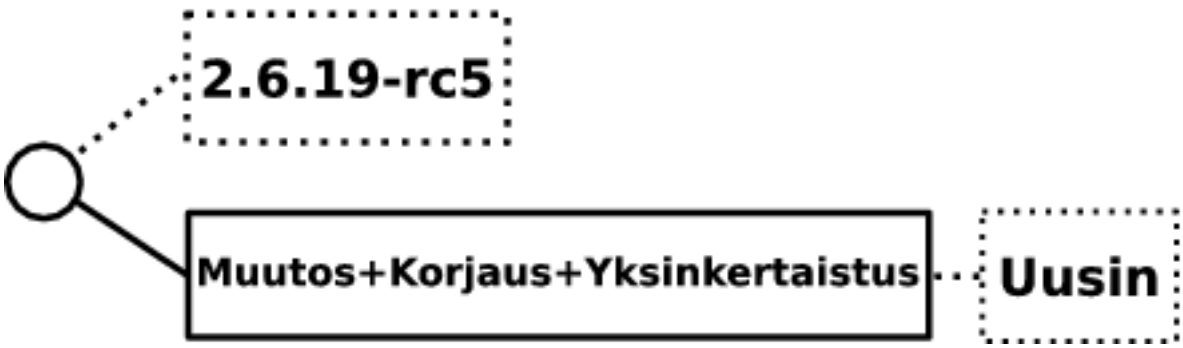
```
If no default mode is specified, it should be grabbed
from the supplied database, not the default one.

[teanropo@jyu.fi: fix it]
[akpm@osdl.org: simplify it]
[akpm@osdl.org: remove pointless DEFAULT_MODEDB_INDEX]
Signed-off-by: Jordan Crouse <jordan.crouse@amd.com>
Cc: Geert Uytterhoeven <geert@linux-m68k.org>
Cc: "Antonino A. Daplas" <adaplas@pol.net>
Signed-off-by: Tero Roponen <teanropo@jyu.fi>
Cc: James Simmons <jsimmons@infradead.org>
Signed-off-by: Andrew Morton <akpm@osdl.org>
Signed-off-by: Linus Torvalds <torvalds@osdl.org>
---

diff --git a/drivers/video/modedb.c b/drivers/video/modedb.c
index d126790..5df41f6 100644
--- a/drivers/video/modedb.c
+++ b/drivers/video/modedb.c
@@ -34,8 +34,6 @@ const char *global_mode_option;
       *  Standard video mode definitions (taken from XFree86)
       */

-#define DEFAULT_MODEDB_INDEX 0
-
 static const struct fb_videomode modedb[] = {
     {
  /* 640x400 @ 70 Hz, 31.5 kHz hsync */
@@ -505,8 +503,10 @@ int fb_find_mode(struct fb_var_screeninfo *var,
  db = modedb;
  dbsize = ARRAY_SIZE(modedb);
     }
+
     if (!default_mode)
-       default_mode = &modedb[DEFAULT_MODEDB_INDEX];
+       default_mode = &db[0];
+
```

```
if (!default_bpp)
   default_bpp = 8;
```



Kuva 4.2: Siistitty kehityshistoria.

Kuten kehityshistoriaa esittävästä kuvasta 4.2 huomataan, muutos on varsin yksinkertainen. Asian taustoja tuntemattomille on siis hyvä kertoa, etteivät asiat oikeasti menneet aivan näin suoraviivaisesti. Jotakin voidaan päätellä jo näistä kommenteista:

```
[teanropo@jyu.fi: fix it]
[akpm@osdl.org: simplify it]
[akpm@osdl.org: remove pointless DEFAULT_MODEDB_INDEX]
```

Katsotaanpa nyt, mitä oikeasti tapahtui.

Linux-ytimen versiossa 2.6.19-rc5 oli seuraavanlainen koodipätkä:

```
if (!default_mode)
   default_mode = &modedb[DEFAULT_MODEDB_INDEX];
```

Koodin tarkoituksena oli asettaa näyttö tilatietokannan määrittelemään oletustilaan, ellei käyttäjä erikseen antanut muuta oletustilaa. Erään kehittäjän tarkoituksena oli muuttaa koodia siten, että tilatietokannan voisi vaihtaa tapauskohtaisesti. Saamansa palautteen perusteella hän päätyi seuraavaan ratkaisuun:

```
if (!default_mode && db != modedb)
   default_mode = &db[0];
else
   default_mode = &modedb[DEFAULT_MODEDB_INDEX];
```

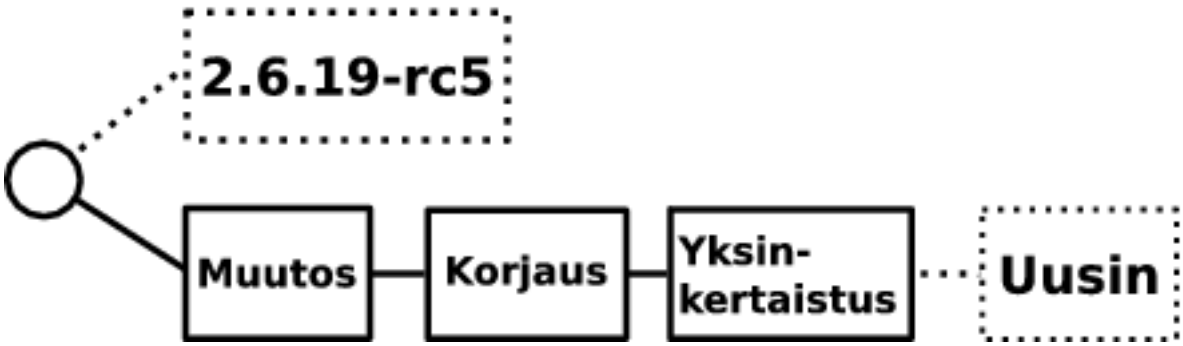
Tämä muutos aiheutti kuitenkin virheen: jos käyttäjä antoi oman oletustilan, mutta ei omaa tilatietokantaa, käyttäjän antama oletustila ylikirjoitettiin. Tämä virhe korjaantui yhdellä lisäehdolla:

```
if (!default_mode && db != modedb)
   default_mode = &db[0];
else if (!default_mode)
   default_mode = &modedb[DEFAULT_MODEDB_INDEX];
```

Linux-ylläpitäjä Andrew Mortonin mukaan saman asian voisi kuitenkin tehdä helpomminkin. Hänen mukaansa yksinkertaisin ratkaisu olisi tällainen:

```
if (!default_mode)
   default_mode = &db[DEFAULT_MODEDB_INDEX];
```

Tätä ratkaisua pidettiin hyvänä, mutta vielä yksi asia nousi esiin: entä jos joku muuttaa koodin käyttämää vakioarvoa? Nopeasti kuitenkin todettiin, ettei kenelläkään ole syytä tehdä sitä, joten koko vakio on turha ja voidaan poistaa.



Kuva 4.3: Todellinen kehityshistoria.

Todellinen kehityshistoria on kuvassa 4.3 esitetyn mukainen. Tässä tilanteessa on kuitenkin yksi ongelmakohta: tehty muutos ei toimi sellaisenaan, vaan vaatii sitä seuraavan korjauksen. Muutos on myös tarpeettoman monimutkainen, joten sitä seuraavan yksinkertaistuksen voisi hyvin tehdä samassa yhteydessä, eikä vasta jälkeenpäin.

Näitä ongelmia ei esiinny lopullisessa versiossa, jossa kaikki tarvittavat muutokset on yhdistetty toisiinsa. Historiaa siistimällä päästään siis edellä esitettyyn lopputulokseen, joka on toimiva ja yksinkertainen, sekä varsin elegantti.

4.25.9 Virheenetsintä

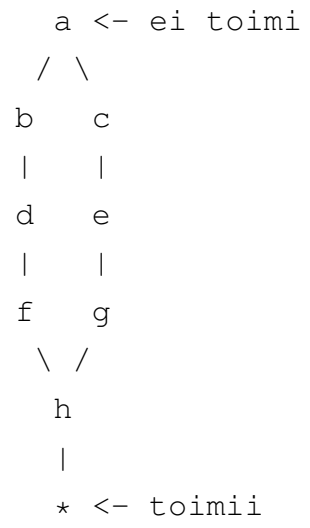
Lähdekoodiin päässeiden virheiden etsintää varten GITiin on kehitetty ns. **bisect**-toiminto, jolla virheitä sisältävät muutokset voidaan löytää muutosten määrän suhteen logaritmisella työmäärällä. Tämä toiminto on tarpeellinen etenkin Linux-ytimen kehityksessä, missä esimerkiksi versioiden 2.6.18 ja 2.6.19 välillä tehtiin jopa 7073 muutosta[161]. Teoriassa virheellisen kohdan pitäisi löytyä näiden joukosta siten, että käyttäjä testaa kaksitoista GITin antamaa konfiguraatiota, mutta virheen laadusta riippuen se löytyy yleensä joko paljon helpommin tai huomattavan paljon vaikeammin. Seuraava esimerkki näyttää, kuinka tämä käytännössä toimii.

1. Aloitetaan etsintä: `git bisect start`.
2. Merkitään jokin tunnetusti toimiva versio: `git bisect good v2.6.0`.
3. Merkitään jokin tunnetusti virheellinen versio: `git bisect bad v2.6.9`.
4. GIT antaa sopivan ehdokkaan, joka merkitään testauksen jälkeen joko toimivaksi tai virheelliseksi: `git bisect bad v2.6.5` tai `git bisect good v2.6.5`.
5. Jos uusia ehdokkaita ei ole enää jäljellä, viimeisin virheelliseksi merkitty ehdokas aiheutti virheen. Muussa tapauksessa palataan edelliseen kohtaan ja testataan uusi ehdokas.

Kuten esimerkistä huomataan, tarkasteltavien muutosten määrä likimäärin puolittuu jokaisella askeleella. Tästä voisi tehdä helposti sellaisen johtopäätöksen, että kyseessä on vain yksinkertainen puolitushaku, jonka voisi tehdä aivan hyvin ilman erillistä komentoakin. Yksinkertaisessa tapauksessa se voisi vielä onnistuakin, mutta asiat muuttuvat, kun mukaan otetaan useita kehityshaaroja ja huomioidaan niiden välillä tehtyjä yhdistämisiä. Tällöin virheellinen muutos voi olla lähtöisin mistä tahansa kehityshaarasta, jolloin sen löytyminen tietystä paikasta ei välttämättä tarkoita sitä, että sen alkuperä olisi selvinnyt.

Otetaan esimerkiksi seuraavanlainen tilanne, joka kuvaa jonkin sovelluksen kehityshistoriaa. Kaaviossa aika kulkee alhaalta ylöspäin, joten on helppo huomata, että muutoksia on tehty samaan aikaan kahdessa eri kehityshaarassa.

[161] Luku on peräisin Greg Kroah-Hartmanin esityksestä Linux Symposium 2007:ssa. http://www.kernel.org/doc/ols/2007/ols2007v1-pages-239-244.pdf (viitattu 24.10.2007).



Kehityshistoriaan on merkitty tähdellä sellainen kohta, jossa tarkastelussa oleva sovellus on toiminut oikein. Tehdyt muutokset on merkitty kirjaimin, ja muutoksen *a* jälkeen on huomattu, ettei sovellus toimi. Varmuudella voidaan sanoa, että virhe aiheutuu jostain muutoksesta *a*–*h*, joten tehtäväksi jää virhekohdan löytäminen tästä joukosta.

Jos kyseessä olisi suoraviivainen, lineaarinen historia, testauskohdan valinta olisi helppo: ajallisesti muutokset *d* ja *e* ovat puolivälissä, joten kumpi tahansa niistä kelpaisi. Todellisuudessa kehitys ei kuitenkaan ole lineaarista, joten valinta pitää tehdä niin, että valinta puolittaa testattavien muutosten määrän mahdollisimman tasaisesti. Aikaa ei pidä ottaa lainkaan huomioon, sillä hajautetussa ympäristössä se ei etene tasaisesti: aikavyöhykkeet, kesä- ja talviajan väliset siirtymät, väärässä ajassa olevat kellot ym. aiheuttavat sen, ettei kahdesta muutoksesta voida suoraan ajan perusteella sanoa, kumpi niistä on uudempi.

Esimerkkitapauksessa sopivia testauskohtia olisivat kohdat *b* ja *c*; toimiva *b* rajaisi kohdat *b*, *d*, *f* ja *h* tarkastelun ulkopuolelle, ja vastaavasti toimivan *c*:n jälkeen olisi selvää, että virheellinen kohta on joko *a*, *b*, *d* tai *f*. Merkittävää tässä on tietenkin se, että tällä tavoin meneteltäessä virhekohta löytyy riippumatta siitä, missä kehityshaarassa se on tehty. Tämä on myös hyvä esimerkki siitä, mihin GITin huima nopeus perustuu: turhaa työtä pyritään välttämään karsimalla epäoleelliset kohdat pois mahdollisimman aikaisessa vaiheessa.

4.25.10 Nykytila

GITin nykytilan selvittelyä varten käytiin läpi kaikki git-listan viestit sen perustamisesta 10.4.2005 aina nykypäivään (30.9.2007) asti. Viestit löytyvät ainakin kahdesta eri arkistosta (http://www.gelato.unsw.edu.au/archives/git/ ja http://marc.info/?l=git), joista jälkimmäistä käytettiin viestimäärien selvittämiseen.

GIT on saavuttanut suuren suosion. Suurimmaksi osaksi tämä johtunee siitä, että sitä käytetään Linux-ytimen kehitykseen, mutta sekään ei riitä selittämään kaikkea. Moni kehitysprojekti on siirtynyt käyttämään GITiä joko virallisesti tai epävirallisesti. Tämä seikka yhdistettynä käyttäjätutkimusten osoittamaan tyytyväisyyteen viittaa siihen, että ohjelma on oikeasti käyttökelpoinen.

Aktiiviset käyttäjä- ja kehittäjäkunnat yhdistettynä ohjelman saamaan julkisuuteen nostavat GITin varteenotettavaksi ehdokkaaksi versionhallintaohjelmien kuninkuusluokkaan. Ohjelman nykytila voidaan kiteyttää seuraaviin X Window System -järjestelmän kehittäjän Keith Packardin sanoihin: "I know Git suffers from its association with the wild and wooly kernel developers, but they've pushed this tool to the limits and it continues to shine. Right now, there's nothing even close in performance, reliability and functionality."

4.26 Mercurial

Mercurial [43, 49][162] on Matt Mackallin keksimään tehokkaaseen *revlog*-tallennusmenetelmään perustuva hajautettu versionhallintaohjelma. Sen kehitys alkoi BitKeeper-kriisin kärjistymisen aikoihin vuonna 2005, ja se oli alun perin tarkoitettu Linux-ytimen kehityksessä käytettäväksi korvaavaksi järjestelmäksi.

4.26.1 Ominaisuudet

Mercurialille asetetut tavoitteet ovat helppokäyttöisyys, nopeus ja skaalautuvuus suurille projekteille. Se on toteutettu Python-ohjelmointikielellä [66], mikä mahdollistaa myös hyvän siirrettävyyden eri käyttöjärjestelmiin. Mercurialiin saa myös uusia ominaisuuksia erillisten lisäosien (engl. extension) avulla, joilla se saadaan mm. tukemaan avainsanojen laajennusta ja aliprojekteja.

Mercurial tallentaa koko projektia koskevat tiedot ns. *manifest*- ja *changeset*-tiedostoihin, joista selviävät mm. projektiin kuuluvat tiedostot ja niiden versiot. Yksit-

[162] http://www.selenic.com/mercurial/ (viitattu 1.8.2007).

täisten tiedostojen tallennukseen Mercurialissa on kehitetty oma *revlog*-menetelmä, missä jokaista tiedostoa kohden pidetään yllä erillistä indeksi- ja datatiedostoa.

Indeksi (engl. revlog index) koostuu 64 tavun vakiomittaisista tietueista, jotka sisältävät tiedoston versioihin liittyviä tietoja, kuten pakkaamattoman version pituuden ja tarkistussumman, sekä tiedon siitä, mihin kohtaan versio sijoittuu datatiedostossa. Kaikki tiedoston versiot tallennetaan pakattuina *datatiedostoon* (engl. revlog data), missä osa pidetään kokonaisina ja osa perusversiosta riippuvina binäärideltoina. Liian pitkiksi kasvavat deltaketjut katkaistaan tallentamalla tarvittaessa uusi versio kokonaisena.

Datatiedostojen ja niihin viittaavien indeksien on tarkoitus olla muuttumattomia. Käytännössä tämä tehdään niin, että niiden loppuun voidaan lisätä tietoa, mutta tätä lisättyä tietoa ei voi enää jälkikäteen muuttaa. Muuttumattomuuden tarkoituksena on mm. ehkäistä tiedostojen pirstaloitumista sekä välttää niiden käsittelyssä syntyviä virheitä. Yksi tästä syntyvä mahdollisuus on myös ns. journaloinnin käyttö, eli viimeisimpiä operaatioita voidaan perua yksinkertaisesti katkaisemalla indeksi- ja datatiedostot oikeasta kohdasta.

Mercurialin dokumentaatio on kattava. Ohjelman käyttöä käsittelevä kirja "Distributed revision control with Mercurial" löytyy verkosta[163] ja se on lisensoitu Open Publication Licensen alaisuuteen. Itse ohjelma käyttää lisenssinään GPLv2:ta.

4.26.2 Toiminta

Mercurial tallentaa koko versiohistorian työhakemiston alle *.hg*-nimiseen alihakemistoon. Komento `hg init` luo kyseisen hakemiston ja täyttää sen tarvitsemillaan tiedoilla.

Lähdekooditiedostojen laitto versionhallintaan tapahtuu kahdessa osassa. Ensin tiedostot merkitään lisättäväksi komennolla `hg add`. Kun kaikki tiedostot on merkitty, ne voidaan tallentaa tietovarastoon komennolla `hg commit`.

Kun tiedosto on versionhallinnassa, sitä voidaan muokata. Muokkauksen voi aloittaa heti, eikä tiedostoja tarvitse lukita. Tehtyjä muutoksia voidaan tarkastella komennon `hg diff` avulla. Kun muutokset on tehty, ne tallennetaan komennon `hg commit` avulla.

Mercurial on muutosjoukkopohjainen, eli usea muutos voidaan tallentaa yhtenä kokonaisuutena. Muutosjoukon tilaa voidaan tarkastella komennolla `hg status`.

[163] http://hgbook.red-bean.com/ (viitattu 14.8.2007).

Versiohistorian tarkasteluun sopii komento `hg log`.

Mercurialissa kehityshaarojen käyttö on yksinkertaista. Uuden kehityshaaran voi aloittaa komennolla `hg branch haara`. Tämän jälkeen kaikki työhakemistossa tehtävät muutokset tallentuvat uuteen haaraan. Olemassaolevien haarojen käyttö tapahtuu päivittämällä työhakemiston sisältö komennon `ht update uusihaara` avulla.

Kehityshaarojen yhdistäminen tapahtuu komennolla `hg merge haara`. Komento yhdistää annetun haaran työhakemiston käyttämään haaraan. Koska yhdistys tapahtuu työhakemiston avulla, koko toiminto pitää päättää normaalien muutosten tavoin komentoon `hg commit`. Olemassaolevien haarojen nimet löytyvät `hg branches` -komennon avulla.

4.26.3 BitKeeper-kriisin vaikutus Mercurialiin

BitKeeper-kriisin vaikutus Mercurialiin on merkittävä, sillä ilman kyseistä tapausta ohjelmaa ei olisi mahdollisesti aloitettukaan. Mercurialin synty ajoittuu päiviin, jolloin tieto BitKeeperin käytön lopettamisesta Linux-kehityksessä oli tullut julkisuuteen. Monien muiden tavoin myös Mercurialin tavoitteena oli täyttää BitKeeperin jälkeensä jättämä aukko. Tässä se olisikin voinut menestyä melko hyvin, ellei Linus Torvalds olisi aloittanut GITin tekoa. Vaikka alkuperäinen tavoite ei toteutunutkaan, Mercurial on pystynyt haalimaan itselleen muita merkittäviä käyttäjiä.

4.26.4 Nykytila

Nykytilan selvittämistä varten käytiin läpi Mercurialin sähköpostiarkiston[164] viestit ajanjaksolta 11.5.2005–30.9.2007.

Ohjelman uusin versio on 0.9.5 lokakuulta 2007. Kehitys jatkuu aktiivisena, joten ohjelman tulevaisuus näyttää valoisalta.

Mercurialin käyttämä Python-kieli on jo yleistynyt sen verran, ettei sen pitäisi olla esteenä ohjelman käyttöönotolle. BitKeeper-kriisin kärjistymisestä alkunsa saaneena ohjelmana Mercurial kilpailee käyttäjistä lähinnä GITin kanssa. Tällä hetkellä Mercurial näyttää jääneen tappiolle, mutta tämä ei kuitenkaan estä sitä olemasta varteenotettava vaihtoehto lähes mille tahansa käytössäolevalle versionhallintaohjelmalle.

Monet projektit, kuten Mozilla ja OpenSolaris, ovat valinneet Mercurialin ver-

[164] http://www.selenic.com/pipermail/mercurial/

sionhallintaohjelmakseen. On mielenkiintoista havaita, että Mercurial näyttäisi olevan suosittu vaihtoehto CVS:n seuraajaksi. Se, miten tämä vaikuttaa Subversionin asemaan, jää nähtäväksi.

4.27 Yhteenveto sovellusten ominaisuuksista

Tutkittujen sovellusten ominaisuudet vaihtelivat ohjelmasta toiseen, mutta sovellusalasta johtuen perusominaisuudet olivat kuitenkin hyvin samankaltaisia. Suurimmat erot johtuivat sellaisista yksittäisten ohjelmien erityisominaisuuksista, joita ei ollut muissa ohjelmissa.

Vanhimmat ohjelmat olivat odotetusti ominaisuuksiltaan rajoitetumpia kuin modernit ohjelmat, mutta nekin täyttivät niille asetetut vaatimukset. Hajautetun kehityksen yleistyminen näkyi ohjelmien tarjoamissa ominaisuuksissa, ja uudemmissa sovelluksissa tietoturvaan oli kiinnitetty entistä enemmän huomiota.

Ohjelmat jakautuivat toteutuksensa perusteella karkeasti kahteen luokkaan. Ensimmäiseen kuuluvat sovellukset rakentuivat jonkin aikaisemman versionhallintaohjelman päälle tai rinnalle, lisäten niihin joitakin toimintoja. Toiseen luokkaan kuuluvat ohjelmat oli toteutettu täysin puhtaalta pöydältä.

Tarkasteltavien sovellusten ominaisuuksia esitellään taulukoissa 4.4 – 4.9. Näitä ja ohjelmaesittelyiden yhteydessä esiintulleita erityisominaisuuksia analysoidaan seuraavassa luvussa.

	Aegis	**Arch**	**ArX**	**Bazaar-NG**
www-sivu	aegis.sourceforge.net/	www.gnu.org/software/gnu-arch/	www.nongnu.org/arx/	bazaar-vcs.org/
lisenssi	GPL v2+	GPL v2+	GPL v2+	GPL v2+
toteutuskieli	C++	C	C++	Python
tallennus-menetelmä	useita vaihtoehtoja	etenevä delta, välimuisti	etenevä delta, välimuisti	knit
yhdistys-menetelmä	riippuu asetuksista	star-merge	star-merge	knit-merge
tietovarasto	hajautettu	hajautettu	hajautettu	hajautettu
käyttöliittymä ja -ympäristö	tekstipohjainen / Unix, cygwin	tekstipohjainen / Unix, osin Windows	tekstipohjainen / Unix, osin Windows	tekstipohjainen / Unix, Windows
atomiset muutokset	kyllä	kyllä	kyllä	kyllä

(jatkuu…)

	Aegis (jatkoa)	**Arch (jatkoa)**	**ArX (jatkoa)**	**Bazaar-NG (jatkoa)**
uudelleen-nimeäminen	eksplisiittinen	eksplisiittinen ja implisiittinen	eksplisiittinen	eksplisiittinen
historian säilyttävä kopiointi	ei	ei	ei	ei
rivikohtainen historia	kyllä	ei	kyllä	kyllä
historian eheyden varmistus	ei	MD5	SHA-256	SHA-1
avainsanojen laajentaminen	ei	ei	ei	ei
pääsynvalvonta	tiedosto-oikeudet	tiedosto-oikeudet	tiedosto-oikeudet	tiedosto-oikeudet, palvelin
aliprojektit	kyllä	osittain	osittain	osittain
merkittävät käyttäjät	—	savannah.gnu.org	—	Ubuntu

Taulukko 4.4: Versionhallintaohjelmien ominaisuuksia (1/6).

	Bky	**Codeville**	**CVS**	**Darcs**
www-sivu	www.triptico.com/software/bky.html	codeville.org/	www.nongnu.org/cvs/	darcs.net/
lisenssi	BSD	BSD	GPL v1+	GPL v2+
toteutuskieli	sh-skripti	Python	C	Haskell
tallennus-menetelmä	kopiointi	limittäinen delta (weave)	takautuva delta	pakatut tiedostot
yhdistys-menetelmä	manuaalinen	Precise Codeville Merge	3-way-merge	Darcs Merge
tietovarasto	hajautettu	hajautettu	keskitetty	hajautettu
käyttöliittymä ja -ympäristö	tekstipohjainen / Unix	tekstipohjainen / Unix, Windows	tekstipohjainen / Unix, Windows, Mac OS	tekstipohjainen / Unix, Windows, OS X
atomiset muutokset	ei	kyllä	ei	kyllä

(jatkuu…)

	Bky (jatkoa)	**Codeville (jatkoa)**	**CVS (jatkoa)**	**Darcs (jatkoa)**
uudelleen-nimeäminen	ei	eksplisiittinen	ei	eksplisiittinen
historian säilyttävä kopiointi	ei	ei	ei	ei
rivikohtainen historia	ei	kyllä	kyllä	kyllä
historian eheyden varmistus	ei	SHA-1	ei	SHA-1
avainsanojen laajentaminen	kyllä	ei	kyllä	ei
pääsynvalvonta	tiedosto-oikeudet	palvelin	palvelin	tiedosto-oikeudet
aliprojektit	ei	ei	kyllä	osittain
merkittävät käyttäjät	—	—	FreeBSD, GNU Emacs, OpenBSD	CL-Debian, DocuWiki, GHC

Taulukko 4.5: Versionhallintaohjelmien ominaisuuksia (2/6).

	DCVS	**FastCST**	**GIT**	**Mercurial**
www-sivu	dcvs.elegosoft.com/	www.zedshaw.com/ projects/fastcst/	git-scm.org/	www.selenic.com/ mercurial/
lisenssi	GPL v1+, osin BSD	GPL v2+, osin Plan9 ja muita	GPL v2	GPL v2
toteutuskieli	C	Ruby	C, sh-skripti, Perl	Python
tallennus-menetelmä	takautuva delta	NST-delta	pakkaus (pack)	revlog
yhdistys-menetelmä	3-way-merge	osittain toimiva	riippuu asetuksista	3-way-merge
tietovarasto	osittain hajautettu	hajautettu	hajautettu	hajautettu
käyttöliittymä ja -ympäristö	tekstipohjainen / Unix, Windows	tekstipohjainen / Unix	tekstipohjainen / Unix	tekstipohjainen / Python-ympäristö
atomiset muutokset	kyllä	kyllä	kyllä	kyllä

(jatkuu…)

	DCVS (jatkoa)	**FastCST (jatkoa)**	**GIT (jatkoa)**	**Mercurial (jatkoa)**
uudelleen-nimeäminen	ei	ei	implisiittinen	eksplisiittinen
historian säilyttävä kopiointi	ei	ei	implisiittinen	kyllä
rivikohtainen historia	kyllä	ei	kyllä	kyllä
historian eheyden varmistus	ei	ei	SHA-1	SHA-1
avainsanojen laajentaminen	kyllä	ei	osittainen	lisäosan avulla
pääsynvalvonta	palvelin	tiedosto-oikeudet, palvelin	tiedosto-oikeudet, palvelin	tiedosto-oikeudet, palvelin
aliprojektit	kyllä	ei	kyllä	lisäosan avulla
merkittävät käyttäjät	elego Software Solutions GmbH	—	Linux, X.Org	Mozilla, OpenSolaris

Taulukko 4.6: Versionhallintaohjelmien ominaisuuksia (3/6).

	Meta-CVS	**Monotone**	**Pastwatch**	**PRCS**
www-sivu	users.footprints.net/ ~kaz/mcvs.html	monotone.ca/	pdos.csail.mit.edu/ pastwatch/	prcs.sourceforge.net/
lisenssi	GPL v2	GPL v2+	GPL v2+	GPL v2+
toteutuskieli	Common Lisp	C++	C++	C++
tallennus-menetelmä	takautuva delta	pakatut tilannekuvat	lohkotut tilannekuvat	vaihteleva
yhdistys-menetelmä	3-way-merge	Mark Merge	3-way-merge	3-way-merge
tietovarasto	keskitetty	hajautettu	hajautettu	paikallinen
käyttöliittymä ja -ympäristö	tekstipohjainen / Unix	tekstipohjainen / Unix, Windows, OS X	tekstipohjainen / Unix, OS X	tekstipohjainen / Unix
atomiset muutokset	ei	kyllä	kyllä	kyllä

(jatkuu…)

	Meta-CVS (jatkoa)	**Monotone (jatkoa)**	**Pastwatch (jatkoa)**	**PRCS (jatkoa)**
uudelleen-nimeäminen	eksplisiittinen ja implisiittinen	eksplisiittinen	eksplisiittinen	eksplisiittinen
historian säilyttävä kopiointi	ei	kyllä	ei	ei
rivikohtainen historia	kyllä	kyllä	ei	ei
historian eheyden varmistus	ei	RSA ja SHA-1	SHA-1	MD5
avainsanojen laajentaminen	kyllä	ei	ei	kyllä
pääsynvalvonta	palvelin	palvelin	palvelin	tiedosto-oikeudet
aliprojektit	kyllä	osittain	ei	kyllä
merkittävät käyttäjät	—	OpenEmbedded	—	—

Taulukko 4.7: Versionhallintaohjelmien ominaisuuksia (4/6).

	RCS	**SCCS/CSSC**	**SourceJammer**	**Subversion**
www-sivu	www.gnu.org/software/rcs/rcs.html	cssc.sourceforge.net/	www.sourcejammer.org/	subversion.tigris.org/
lisenssi	GPL v2+	CDDL v1.0 / GPL v2+	GPL v2+ / LGPL v2.1+	Apache / BSD-lisenssin kaltainen
toteutuskieli	C	C++	Java	C
tallennus-menetelmä	takautuva delta	limittäinen delta (weave)	kopiointi	hyppydelta
yhdistys-menetelmä	3-way-merge	—	3-way-merge	3-way-merge
tietovarasto	paikallinen	paikallinen	keskitetty	keskitetty
käyttöliittymä ja -ympäristö	tekstipohjainen / DOS, OS/2, Unix	tekstipohjainen / Unix, Windows, OS X	graafinen / Java-ympäristö	tekstipohjainen / Unix, Windows, OS X
atomiset muutokset	—	—	ei	kyllä

(jatkuu…)

	RCS (jatkoa)	**SCCS/CSSC (jatkoa)**	**SourceJammer (jatkoa)**	**Subversion (jatkoa)**
uudelleen-nimeäminen	—	—	eksplisiittinen	eksplisiittinen
historian säilyttävä kopiointi	—	—	kyllä	kyllä
rivikohtainen historia	ei	kyllä	ei	kyllä
historian eheyden varmistus	ei	heikko tarkistussumma	ei	MD5
avainsanojen laajentaminen	kyllä	kyllä	kyllä	kyllä
pääsynvalvonta	tiedosto-oikeudet	tiedosto-oikeudet	palvelin	palvelin
aliprojektit	ei	ei	kyllä	kyllä
merkittävät käyttäjät	—	—	—	Apache, GCC, GNOME, KDE, ReactOS, Samba

Taulukko 4.8: Versionhallintaohjelmien ominaisuuksia (5/6).

	Superversion	**SVK**	**Vesta**	
www-sivu	www.superversion.org/	svk.elixus.org/	www.vestasys.org/	
lisenssi	GPL v1+ ja muita	Artistic v1 / GPL v2	LGPL v2.1+	
toteutuskieli	Java	Perl	C++	
tallennus-menetelmä	deltat, tietokanta	hyppydelta	kopiointi	
yhdistys-menetelmä	3-way-merge	smerge	Precise Codeville Merge	
tietovarasto	keskitetty	hajautettu	hajautettu	
käyttöliittymä ja -ympäristö	graafinen / Java-ympäristö	tekstipohjainen / Unix, Windows, OS X	tekstipohjainen / Unix	
atomiset muutokset	kyllä	kyllä	kyllä	

(jatkuu…)

	Superversion (jatkoa)	**SVK (jatkoa)**	**Vesta (jatkoa)**	
uudelleen-nimeäminen	ei	eksplisiittinen	eksplisiittinen	
historian säilyttävä kopiointi	ei	kyllä	kyllä	
rivikohtainen historia	ei	kyllä	ei	
historian eheyden varmistus	oma algoritmi	MD5	oma algoritmi	
avainsanojen laajentaminen	kyllä	kyllä	ei	
pääsynvalvonta	tiedosto-oikeudet, palvelin	palvelin	palvelin	
aliprojektit	ei	kyllä	ei	
merkittävät käyttäjät	Project Prophecy	Best Practical	Intel	

Taulukko 4.9: Versionhallintaohjelmien ominaisuuksia (6/6).

5 Analyysi

> *Although several well-understood solutions are available, no single SCM system offers all solutions at once. Integration and flexibility are thus still issues for SCM users and SCM vendors—maybe also for SCM researchers, provided they find a way to validate the practical benefits of new SCM models.*
>
> *— Karol Frühauf & Andreas Zeller [19]*

Aiemmissa luvuissa käytiin läpi versionhallintaan liittyviä käsitteitä, ja esiteltiin aiheeseen liittyvää tutkimusta ja tärkeimpiä algoritmeja. Näiden jälkeen tutustuttiin avoimen lähdekoodin liikkeeseen ja seurattiin versionhallintaohjelmien historiaa aina 1970-luvun alkupuolelta 2000-luvun puoliväliin. Seuraavaksi osoitettiin, kuinka versionhallintaohjelmat ovat kehittyneet avoimen lähdekoodin puolella, ja nostettiin esiin sellaisia innovaatioita, joita ei ole aiemmin tutkittu, tai jotka osoittavat aiemmat johtopäätökset vanhentuneiksi ja uudelleenarviointia kaipaaviksi.

Tässä luvussa tarkastellaan edellisten lukujen sisältöä analyyttisestä näkökulmasta ja katsotaan, minkälaisia johtopäätöksiä niistä voidaan tehdä. Ensin analysoidaan tutkimusongelmaa, -menetelmää ja -aineistoa, minkä jälkeen arvioidaan lyhyesti ohjelmien yleisiä ominaisuuksia. Seuraavaksi tarkastellaan löytyneitä erikoistoimintoja ja arvioidaan niiden tieteellistä merkittävyyttä. Lopuksi esitetään mahdollisia jatkotutkimusaiheita.

5.1 Tutkimuksen toteutus

Tutkielman tavoitteena oli selvittää, onko avoimen lähdekoodin versionhallintaohjelmien kehityksessä syntynyt sellaisia innovaatioita, joilla on myös tieteellistä arvoa. Tämän päätavoitteen lisäksi toisena tavoitteena oli tutustua mahdollisimman moneen avoimeen versionhallintaohjelmaan, ja selvitellä niiden toimintaa käytännössä.

Avoimen lähdekoodin merkitys ohjelmistoteollisuudelle on kiistaton. Moni tutkimus on osoittanut avoimen lähdekoodin sovellusten olevan sekä laadukkaita että toimivia. Sen sijaan tieteellinen vaikutus on jäänyt lähes kokonaan tutkimuksen ulkopuolelle. Tämä tutkielma pyrkii täyttämään osaltaan tätä aukkoa.

Tutkimuksen hypoteesinä oli, että avoimen lähdekoodin versionhallintaohjelmien kehityksessä on saatu aikaan tieteellisesti merkittäviä tuloksia, joita ei ole tuotu esille. Tämän seikan paikkansapitävyyttä ei ollut mahdollista tietää ennen tutkimuksen aloittamista, sillä kummankin vaihtoehdon perustelu olisi vaatinut todisteita, joita ei niiden luonteesta johtuen ollut löydettävissä.

5.1.1 Tutkimusaineisto

Tutkimuskohteeksi valittiin joukko avoimen lähdekoodin versionhallintaohjelmia. Lähes jokaisesta tarkasteltavasta sovelluksesta on laadittu jonkinlainen julkaisu, joskin niiden tieteellinen taso vaihtelee asiantuntijoiden laatimista käyttöoppaista aina väitöskirjaan saakka. Esimerkkejä näistä julkaisuista ovat mm. [46] (Aegis), [4] (CVS), [24] (GIT), [43] (Mercurial), [29] (Monotone), [7] (Pastwatch), [42] (PRCS), [65] (RCS), [57] (SCCS), [8] (Subversion) ja [27] (Vesta), joista osalla on nykyään jopa historiallista arvoa.

Julkaisujen lisäksi aineistoa hankittiin eri sovellusten kehityskeskusteluissa käytettävien sähköpostilistojen viestiarkistoista. Omat vaikeutensa tähän toi heti alussa tehty havainto siitä, ettei viestien otsikoista voinut useinkaan päätellä niiden sisältöä. Etenkin kovasti rönsyilevissä keskusteluissa vanhan otsikon alla voitiin esittää lähes mitä tahansa asioita, jotka olisivat jääneet huomaamatta, ellei kaikkia viestejä olisi käyty järjestelmällisesti läpi.

Sähköpostilistojen lisäksi käytössä oli myös muita kommunikointivälineitä, kuten IRC-kanavia. Nämä kuitenkin rajattiin tutkimuksen ulkopuolelle, koska niiden käyttö olisi voinut johtaa tutkimusaineiston muuttumiseen reaaliaikaisuuden vuoksi. Rajaukseen vaikutti myös von Hippelin ym. [68] muistutus, ettei kaikki kommunikointi ole aina julkista.

Valmiin materiaalin lisäksi joissakin tapauksissa perehdyttiin myös sovelluksen lähdekoodiin sekä tehtiin omia testejä, joilla pyrittiin varmistamaan tietojen oikeellisuus. Joitakin sovelluksia kokeiltiin myös käytännössä.

5.1.2 Aineiston valinta

Tutkimusaineiston valintaan vaikutti monta seikkaa. Koska tavoitteena oli etsiä piiloon jääneitä innovaatioita, niistä ei ollut mahdollista löytää valmiita julkaisuja. Tämän vuoksi tutkimuksessa pyrittiin pääsemään ideoiden alkulähteille, eli paikkoihin, joissa ne tuodaan esille ensimmäisen kerran. Sähköpostilistojen lisäksi tällaisik-

si osoittautuivat myös sovellusten lähdekoodin TODO-merkinnät, joiden merkitykseen myös Lin [40] kiinnittää huomiota.

Ohjelmien dokumentaatio on kirjoitettu käyttäjiä varten. Tämän vuoksi ne ovat yleensä täynnä selostusta ohjelman hienouksista ja eduista kilpailijoihin nähden. Kaikesta huolimatta käyttöohjeet antavat yleiskuvan ohjelman toiminnallisuudesta, joten niiden avulla voi arvioida ohjelman käyttäjäystävällisyyttä.

Aineiston monimuotoisuudesta johtuen myös sen laatu vaihteli huomattavasti. Joidenkin ohjelmien kohdalla dokumentaatio oli täysin julkaisukelpoista, ja toisten kohdalla se puuttui kokonaan. Tämän epätasaisuuden vaikutusta ohjelmien arvioinnissa kompensoitiin selvittämällä puuttuvia tietoja verkkolähteistä (Google), mutta sen ei annettu vaikuttaa ohjelmasta saatuun kuvaan.

Tutkimusaineisto oli erittäin käytännönläheistä. Suuri osa tutkituista sähköpostiviesteistä käsitteli ohjelman kehitystä, mutta mukana oli myös käyttäjien opastusta ongelmatapauksissa. Sähköpostiarkistojen viestimääristä pystyi hahmottamaan ohjelman käyttäjämäärää, mutta roskapostin osuutta oli vaikea arvioida.

Käytetty aineisto takasi tietojen oikeellisuuden eri ohjelmien kohdalla, mutta kilpailevia ohjelmia käsittelevistä viesteistä huomasi lievää aliarvostusta. Tämä näkemysten värittyneisyys oli kuitenkin ennakoitua, joten se osattiin ottaa huomioon, eikä se siten päässyt vaikuttamaan asiatietojen oikeellisuuteen. Sähköpostiarkistojen käytön suurin hyöty oli siinä, että ne mahdollistivat käsiteltyjen aiheiden tarkastelun oikeassa historiallisessa kontekstissa.

Koska käytetty aineisto on täysin julkista, tutkimus on tarvittaessa mahdollista uusia, ja verrata näin saatuja tuloksia tämän tutkielman tuloksiin. Tällöin on tosin muistettava ottaa huomioon se seikka, että tutkimuskohde muuttuu koko ajan. Vaikka sähköpostiarkistot ovatkin pysyviä, www-sivujen sisällöt voivat muuttua hyvinkin nopeasti. Jonkin verran asiaa auttavat erilaiset www-arkistot, mutta nekään eivät ratkaise tätä ongelmaa täydellisesti.

5.1.3 Tutkimusmenetelmä

Tutkimusmenetelmäksi valittiin vertaileva empiirinen tutkimus. Tutkimusaineistossa tiedettiin olevan ristiriitaisia näkemyksiä, joten näiden käsittelyssä päädyttiin käyttämään Pengin [50] esittämää kiinalaista dialektiikkaa. Käytännössä tämä tarkoitti ristiriitojen selvittämistä ottamalla huomioon se konteksti, missä nämä esiintyivät.

Käytettyä menetelmää ei ehkä tunneta kovin hyvin ainakaan nimeltä, eikä sitä

ole juurikaan aiemmin sovellettu tietotekniikan alalla. Se on kuitenkin erinomainen menetelmä tilanteisiin, joissa olosuhteet vaihtuvat nopeasti, eikä erilaisten näkemysten ratkaisuun ole mahdollista valita vain tiettyä vaihtoehtoa. Tällaisia tilanteita esiintyy etenkin tietotekniikassa, jossa eilisen parhaaksi tiedetty sovellus voi tänään olla vain keskinkertainen.

Empiiristen ja havainnoivien tutkimusmenetelmien yhtenä ongelmana pidetään sitä, että tutkijan osallistuminen ja havaintojen teko jo itsessään saattavat vaikuttaa tutkittavaan ilmiöön, jolloin saadut tulokset eivät välttämättä vastaa normaalia tilannetta. Tämän tutkielman tekoon tällaista vaaraa ei kuitenkaan liittynyt, sillä tarkasteltava aineisto oli historiallista ja liittyi sellaisiin jo menneisiin tapahtumiin, joihin ei voinut enää vaikuttaa.

Käytetty havainnoiva ja kokeileva tutkimusmenetelmä osoittautui melko työlääksi. Parityönä tehtynäkään tutkimuksessa ei ehditty perehtyä kaikkien käsiteltävien sovellusten historiaan kuin muutaman vuoden taakse nykyhetkestä. Tätä varhaisemmat tiedot piti hankkia kirjallisuudesta, joten niiltä osin omien havaintojen teko jäi melko vähäiseksi. Tutkielmassa tämä näkyy siinä, että vanhempia sovelluksia käsitellään yleensä tiukasti lähdekirjallisuuden valossa, mutta uudempien kohdalla voidaan viitata myös asioita käsitteleviin alkuperäisiin sähköpostiviesteihin.

Joitakin sovelluksissa mainostettuja ominaisuuksia testattiin myös käytännössä. Testien perusteella ohjelmista annetut tiedot vaikuttivat pitävän hyvin paikkansa, eikä suoranaisia virheellisyyksiä löytynyt mistään.

Tutkielman suurin virhemahdollisuus liittyy käytettyyn tutkimusmenetelmään. Dialektista menetelmää käytettäessä tutkija tekee omia johtopäätöksiä ja voi itse määritellä, minkälaisen painoarvon antaa millekin lähteelle. Tämän vuoksi tutkijan omat ennakkoluulot ja -käsitykset voivat ohjata tulosta “haluttuun” suuntaan, ellei niitä osata sulkea pois. Täydellinen puolueettomuus on kuitenkin niin vaikea saavuttaa, että tuloksia voidaan pitää aina subjektiivisina.

Edellä esitettyä ei pidä käsittää väärin niin, että koko menetelmä olisi käyttökelvoton. Subjektiivisuus koskee lähinnä aineiston valintaa ja siitä tehtäviä tulkintoja, mutta ei salli tulosten tahallista vääristelyä. Käytännössä erimielisyyksiä voi tulla vaikkapa siitä, onko jonkin operaation kymmenen sekunnin kesto tutkijan johtopäätösten mukaisesti “erittäin hyvä aika”, vai jotain muuta, joten mistään vakavasta ongelmasta ei ole kyse.

Tämän tutkielman sisältöön mahdollisesti vaikuttaneita tekijöitä ovat mm. kirjoittajien ohjelmointitausta ja aiempi kokemus joistakin tutkituista sovelluksista. Edel-

lisen perusteella asioita ei ole ehkä osattu katsoa peruskäyttäjän näkökulmasta, joten huomio on kiinnittynyt liiaksi teknisten asioiden esittelyyn. Jälkimmäinen on puolestaan voinut antaa "tutuille" sovelluksille jonkin verran etua, sillä niistä on osattu nostaa esiin sellaisia yksityiskohtia, jotka eivät ole yleisessä tiedossa.

5.1.4 Aikaisempi tutkimus

Ohjelmiston konfiguraation- ja versionhallinnan merkitystä sekä niiden yhteydessä esiintyviä merkittäviä ideoita on aiemmin käsitelty mm. Estublierin ym. artikkelissa *Impact of Software Engineering Research on the Practice of Software Configuration Management* [16] vuodelta 2005. Siinä tarkastelu kuitenkin rajoitetaan tieteellisen ja teollisen tutkimuksen saavutuksiin, joten avoimen lähdekoodin puolella tehdyt keksinnöt jäävät käsittelyn ulkopuolelle. Tämän tutkielman kannalta kyseisen tutkimuksen heikoimpana kohtana voidaankin pitää seuraavaa oletusta [16]:

> Of course, there is a chance of some unpublished invention being heavily used in some real product. However, this is unlikely, because competition among the SCM vendors forces the major players to offer comparable functionality and feature sets. [...] We concluded that basing this report on research published in the open literature would be adequate.

Oletus, jonka mukaan kaikista merkittävistä keksinnöistä tiedotetaan asianmukaisesti, ei päde suurimpaan osaan avoimen lähdekoodin projekteista. Niissä tavoitteena ei ole saada suosiota vaan ainoastaan parannella oman sovelluksen toimintaa. Julkisen kehityksensä ansiosta niihin ei myöskään liity huolta julkaisemattomien keksintöjen unohtumisesta, sillä ne ja yleensä niiden koko kehityshistoria ovat koko ajan kaikkien hyödynnettävissä. Näiden piirteiden vuoksi tehty oletus ei päde avoimen lähdekoodin puolella, joten niiden rajaaminen käsittelyn ulkopuolelle on perusteltua.

Edellä mainitun Estublierin ym. artikkelin ulkopuolista avoimen lähdekoodin versionhallintaa käsittelevät mm. Asklundin ym. artikkeli *Configuration Management for Open Source Software* [3] vuodelta 2001 ja Robertin esitys *DVCS or a new way to use Version Control Systems for FreeBSD* [56] vuodelta 2006. Aihetta sivuaa myös Thomas Kellerin kandidaatintutkielma *Open Source Version Control* [37] vuodelta 2006, jossa käsitellään versionhallinnan käytänteiden lisäksi myös avoimen lähdekoodin

historiaa ja verrataan lyhyesti kolmea eri versionhallintaohjelmaa. Saatavilla olevan aineiston perusteella onkin helppo todentaa Löhin ym. [41] väite, jonka mukaan versionhallintaa ei tieteellisessä kirjallisuudessa käsitellä enää itsenäisenä alueena, vaan lähes aina jonkin laajemman kokonaisuuden osana.

5.2 Versionhallintaohjelmien kehitys käytännössä

Tutkielman alkupuolella (luku 3.4 s. 50) esiteltiin erilaisia näkemyksiä avoimen lähdekoodin käsitteen merkityksestä. Joidenkin käsitysten mukaan avoimuus riippuu käytettävästä kehitysmenetelmästä, kun se toisten mukaan on vain lisenssikysymys. Tässä tutkielmassa tarkasteltavilta sovelluksilta avoimuuden osoitukseksi riitti jonkin avoimen lähdekoodin lisenssin käyttö, joten nyt on hyvä tarkastella vielä niiden kehitysmenetelmiä.

5.2.1 Katedraali ja basaari

Esiteltyjen sovellusten taustat ja sen myötä niiden kehitystavat vaihtelevat melkoisesti. Esimerkiksi Vesta on suuryrityksen omaan käyttöönsä toteuttama raskaan sarjan konfiguraationhallintaohjelma, ja Bky puolestaan yhden hengen ylläpitämä komentotiedosto. Näiden ääripäiden väliin mahtuukin paljon erilaisia sovelluksia, joiden ainoana yhteisenä piirteenä voidaan avoimuuden lisäksi pitää sitä, että ne soveltuvat jonkinlaiseen versionhallintaan.

Vaikka Eric S. Raymondin ajatuksia avoimen lähdekoodin kehitysmenetelmistä ei aina pidetäkään arvossa, niiden merkitystä ei pidä silti aliarvioida. Raymondin jaottelun mukaisia eroja katedraali- ja basaari-tyylisen kehityksen välillä on selvästi havaittavissa eri projektien välillä.

Linus Torvaldsin GITissä käytetään odotetusti basaari-tyylistä kehitystapaa puhtaimmillaan. Se ei olekaan mikään ihme, sillä siinä käytettävä tapa on sama kuin Linux-ytimessä, josta Raymond sai alkuperäisen inspiraationsa. Selvästi katedraalityylisintä kehitysmallia käytettiin puolestaan Tom Lordin epäonnistuneessa yrityksessä Arch 2:n (revc, 4.12.4, s. 103) suhteen. Lordin laatimista täsmällisistä suunnitelmista huolimatta projekti ei herättänyt riittävästi mielenkiintoa, joten se jäi lyhytikäiseksi.

Suuri osa sovelluksista oli korkeintaan muutaman henkilön kehittämiä, eikä niiden ympärille ollut muodostunut suurta käyttäjäkuntaa. Tämä johtunee siitä, et-

tei ohjelmia ole alun perinkään tarkoitettu kovin suureen käyttöön, vaan koodi on julkaistu “sellaisenaan”. Tällaiset ohjelmat eivät käytä basaari-tyyliä sen koommin kuin katedraali-tyyliäkään.

Vesta poikkesi muista tutkituista ohjelmista siinä, että sen siirtyminen avoimen lähdekoodin ohjelmaksi ei ollut aivan kitkatonta. Erityisesti ongelmaksi muodostui ohjelman käyttöönotto, sillä sen lähdekoodista sai käännettyä ajokelpoisen sovelluksen vain Vestaa itseään käyttämällä, eli avoimen lähdekoodin hyödyntäminen vaati sovelluksen binääriversion käyttöä. Tätä ei aluksi osattu pitää merkittävänä ongelmana, mutta myöhemmin selvisi, että etenkään monet vapaiden ohjelmien kannattajat eivät suostu käyttämään ohjelmia, joiden lähdekoodia eivät ole voineet tarkastaa.

Vaikka asia korjattiin myöhemmin tarjoamalla Make-pohjainen kääntömahdollisuus, on selvää, että huonon ensivaikutelman vuoksi Vesta menetti osan potentiaalisista käyttäjistään.

5.2.2 Tutkimustulosten hyödyntäminen

Tieteellisten tutkimustulosten hyödyntäminen vaihteli eri ohjelmien kesken. Yksinkertaisimmat ohjelmat, kuten Bky, perustuivat olemassa oleviin sovelluksiin, kuten *patch*:iin ja *diff*:iin. Näitä käytettäessä perustoiminnallisuus voitiin toteuttaa ilman tietoa teoreettisista perusteista. Toista ääripäätä edustivat ohjelmat, kuten RCS, joiden toiminta perustui kehittäjien julkaisemiin tieteellisiin löydöksiin.

Ääripäiden välissä suhtautuminen tutkimustuloksiin oli kirjavaa. Monissa sovelluksissa käytettiin tieteellisesti tunnettujen algoritmien paranneltuja versioita, ja toiset käyttivät alusta alkaen itse tehtyjä toimintoja. Yhteistä näille tavoille oli, ettei parannuksia tai ideoita pyritty tuomaan esille tieteellisinä. Ainoat käytetyt kanavat tulosten julkaisuun olivat erilaiset konferenssit, joissa ohjelmia esiteltiin, mutta silloinkin pääasiassa olivat itse ohjelmat, eivät niiden käyttämät algoritmit.

Tutkimustulosten käsittelyssä yksi aihe nousi ylitse muiden. Kun MD5- ja SHA-1-algoritmeistä julkaistiin artikkelit, joissa kerrottiin uusista saavutuksista niiden murtamisessa, keskustelu sähköpostilistoilla vilkastui. Tämä johtui kyseisten algoritmien käytöstä eri ohjelmissa, mutta osoitti myös, miten tieteen saavutuksia seurataan aktiivisesti.

Eräs mielenkiintoinen havainto oli, että muista algoritmeista poiketen kryptografiset toiminnot perustuivat aina tieteellisiin tuloksiin, eikä mikään ohjelma pyrkinyt käyttämään niiden tilalla omia keksintöjä. Tämä viittaa kahteen vaihtoehtoon.

Joko kryptografian tieteellinen tutkimus on niin pitkällä, ettei parannettavaa enää löydy, tai versionhallintaohjelmien kehittäjiltä puuttuu aiheeseen vaadittava osaaminen, mikä johtaa valmiiden ratkaisujen käyttöön. Voidaan kuitenkin sanoa, että ainakin tässä asiassa tiede on käytännön tasalla.

5.3 Havaintoja versionhallintaohjelmista

Kaikki tutkimuskohteet eivät osoittautuneet merkittäviksi. Suuri osa tutkituista versionhallintaohjelmista osoittautui lähemmissä tarkasteluissa perinteisiä menetelmiä käyttäviksi retrotuotteiksi, joista ei juurikaan löytynyt innovaatioita. Näistä pienistä pettymyksistä huolimatta tutkimusta voidaan pitää onnistuneena, sillä muutamasta sovelluksesta löytyi sellaisia ominaisuuksia, joista voidaan käyttää nimitystä innovaatio.

Kuten lähes aina, myös versionhallinnan yhteydessä uudet ideat perustuvat aikaisemmille tuloksille. Avoimen lähdekoodin projekteissa uusia asioita tehdään yleensä vasta todellisen tarpeen mukaan, koska ne eivät välttämättä edes toimisi liian aikaisessa vaiheessa.

On helppo ymmärtää, ettei kehityshistorian siistimistä kannata edes harkita, ennen kuin jokaisella kehittäjällä on oma tietovarasto. Yhteisen historian muuttaminen ei tule kysymykseenkään, koska se vaikuttaisi kerralla kaikkiin muihinkin kehittäjiin. Samalla tavoin voidaan ajatella, että verkko-ominaisuuksien laittaminen SCCS:ään 1970-luvulla olisi ollut ennenaikaista, mutta 1990-luvun CVS:ään ne sopivat jo varsin luontevasti. Verkko-ominaisuuksien puuttuminen 2000-luvun versionhallintaohjelmista olisikin jo puolestaan vakava puute.

Tämän esimerkin perusteella voidaan huomata, että jotkut uusina esitetyt innovaatiot voivat olla sellaisia, että niitä on joskus aiemmin harkittu, mutta ne eivät ole olleet silloin käyttökelpoisia. Tämä vastaa tilannetta, jossa joku on aikoinaan todistanut, että X ei ole toimiva idea, ja myöhemmät tutkijasukupolvet ovat näihin todistuksiin vedoten jättäneet idean huomiotta. Myöhemmin joku aikaisempiin tutkimuksiin perehtymätön ohjelmoija voi keksiä saman idean itse, jolloin se osoittautuukin toimivaksi. Tämän vuoksi onkin joskus hyvä kyseenalaistaa vanhoja tutkimustuloksia ja katsoa, ovatko ne vielä asianmukaisia.

Seuraavaksi arvioidaan tutkituista ohjelmista löydettyjä yhteisiä ominaisuuksia. Erikoisominaisuudet käsitellään myöhemmin.

5.3.1 Lisenssi

Kaikki tarkastellut sovellukset olivat jo valintaperusteidensa mukaisesti jonkin avoimen lähdekoodin lisenssin alaisia, minkä voikin selvästi huomata taulukosta 5.1, johon on koottu tiedot käytettyjen lisenssien saatavuudesta.

Lisenssi	**Lisenssiteksti verkossa**
Apache	http://www.opensource.org/licenses/apachepl.php
Artistic	http://www.opensource.org/licenses/artistic-license.php
BSD	http://www.opensource.org/licenses/bsd-license.php
CDDL	http://www.opensource.org/licenses/cddl1.php
GPL	http://www.opensource.org/licenses/gpl-license.php
LGPL	http://www.opensource.org/licenses/lgpl-license.php
Plan 9	http://www.opensource.org/licenses/plan9.php

Taulukko 5.1: Lisenssien verkko-osoitteet.

Selvästi käytetyin lisenssi oli GNU General Public License (GPL), josta oli käytössä useita eri versioita. Eniten käytettiin versiota 2, mutta joukossa oli myös muutama alkuperäisen GPLv1:n alainen sovellus (CVS ja DCVS). Lisenssin 29. kesäkuuta 2007 julkaistua kolmosversiota ei käytetty missään tutkituista sovelluksista, mikä johtunee kyseisen lisenssin uutuudesta. On kuitenkin hyvin todennäköistä, että monet GPL:n alaiset ohjelmat päivittävät käyttämänsä lisenssin uudempaan tulevien julkaisujen yhteydessä.

Eräs mielenkiintoinen lisensseihin liittyvä havainto oli, että lähes kaikkien sovellusten www-sivuilta puuttui tieto siitä, mikä lisenssin versio oli käytössä. Joukkoon mahtui myös tapaus, missä sovelluksen todellinen lisenssi poikkesi sen sivustolla ilmoitetusta. Näistä voisi varovaisesti päätellä sen, ettei lisenssin sisältöön juurikaan kiinnitetä huomiota. Monille riittää, että käyttöön otetaan jokin tunnettu avoimen lähdekoodin lisenssi.

5.3.2 Toteutuskieli

Avointa lähdekoodia sosiologian näkökulmasta käsittelevässä väitöskirjassaan Yuwei Lin antaa ohjelmointikielille varsin suuren merkityksen [40]:

> Programming languages, which consist of symbolic contents, should be seen not merely as algorithm codes but also as social codes. Program-

> ming languages or software tools, materialised in versatile forms, sometimes could be indication of users' identities. When one chooses to code with EMACS, s/he might try to show her/his sympathy with Stallman's philosophy.

Tässä tutkielmassa tarkasteltujen sovellusten toteutuskielten joukko oli kirjava, joten mukaan oli päässyt mm. shelliskripteillä, C-kielellä, Pythonilla, Haskellilla, Rubyllä, Javalla ja Common Lispillä toteutettuja sovelluksia.

Käytettyjen kielten valintaan oli ollut monia syitä. Vanhimmat ohjelmat oli kirjoitettu kielillä, jotka olivat käytössä niiden aloitusaikoina (esimerkiksi SNOBOL4), ja vastaavasti uudemmissa sovelluksissa oli käytetty nykyisin suosittuja kieliä. Tämä ei ollut kuitenkaan mikään ehdoton periaate, sillä joukosta löytyi myös monia poikkeuksia.

Ohjelmat, jotka rakentuivat jonkin toisen versionhallintaohjelman päälle, käyttivät yleensä eri toteutuskieltä, kuin perustana toimiva ohjelma. Esimerkiksi Meta-CVS rakentui CVS:n päälle ja oli toteutettu Common Lisp:llä, kun taas CVS oli kirjoitettu C-kielellä. SVK puolestaan oli toteutettu Perlillä ja sen käyttämä Subversion C++:lla. Poikkeuksen teki DCVS, joka oli toteutettu C-kielellä, kuten sen käyttämä CVS:kin. Tämä tosin johtui siitä, että DCVS käytti osittain samaa lähdekoodia kuin CVS.

Erilaisten toteutuskielten käyttö johti erilaisiin riippuvuuksiin. Esimerkiksi Meta-CVS:n käyttö vaati GNU CLISP -ohjelman, joka on eräs Common Lisp -kielen toteutus. On kuitenkin epätodennäköistä, että kyseinen ohjelma olisi asennettu tavallisille käyttäjille oletuksena, joten sen vaatiminen saattaa nostaa myös Meta-CVS:n käyttökynnystä. Mercurial puolestaan käytti Python-kieltä, mikä löytyykin jo useammalta koneelta. C-kieliset ohjelmat eivät vaadi juuri ylimääräisiä riippuvuuksia, mikä antaa niille jonkin verran etua.

Versionhallintaohjelmiin keskittyvän BetterSCM-sivuston[1] ylläpitäjä Shlomi Fish kritisoi Common Lispin käyttöä Meta-CVS:n toteutukseen seuraavasti[2]:

> Another downside to Meta-CVS is that it is written in Common LISP. [...] If Meta-CVS' author wishes to make it more popular, I strongly advise him to re-implement it in C, Perl, Python or something more standard.

[1] http://better-scm.berlios.de/ (viitattu 25.10.2007).

[2] http://better-scm.berlios.de/docs/nice_trys.html (viitattu 25.10.2007).

Tämä kritiikki ei ollut kuitenkaan ainutlaatuista, sillä samalla sivustolla Fish kritisoi myös Darcsin Haskellin käyttöä. Robert [56] puolestaan pohtii Mercurialin mahdollisuuksia FreeBSD-projektissa todeten, että se vaatisi joko ulkoisen Pythonin käyttöä tai sen sisällyttämisen projektin ydinsovelluksiin.

Vertailussa mukana olleista ohjelmista parhaiten menestyneet ohjelmat oli toteutettu perinteisillä ja tunnetuilla ohjelmointikielillä. Oudompia kieliä käyttäneet sovellukset olivat melko vähäisessä käytössä, mikä nostaa esille mahdollisuuden, että käytetyllä ohjelmointikielellä ja ohjelman saamalla suosiolla on jonkinlainen yhteys. Tätä mahdollisuutta tukee myös Linin [40] havainto, jonka mukaan ohjelmointikielillä on tärkeä osa ohjelmointikulttuurien muotoutumisessa ja niiden välisten rajojen asettumisessa.

5.3.3 Tallennusmenetelmä

Versionhallinnassa on lähes aina kiinnitetty huomiota käytettävän tallennustilan minimointiin, ja etenkin alkuaikoina se oli jopa koko versionhallinnan päätavoite. Tilansäästöön pyritään vielä nykyäänkin useimmiten erilaisia delta-menetelmiä soveltamalla, joten sen suhteen edistys on näkynyt lähinnä algoritmien kehittymisenä.

Tutkitut sovellukset jakautuivat käyttämänsä tallennusmenetelmän mukaan karkeasti kahteen luokkaan, kokonaiset tiedostot tallentaviin ja deltoja käyttäviin. Poikkeuksena tästä on PRCS, jonka tallennusmenetelmä on tarkoituksella jätetty määrittelemättömäksi, joten se voi vaihdella käytetystä ohjelmaversiosta riippuen.

Vanhimmat versionhallintaohjelmat, kuten SCCS ja RCS, tallensivat muutokset odotetusti deltoina, sillä aktiiviaikoinaan kummankin tärkeänä tehtävänä oli tilansäästö. Nykyään tallennuskapasiteetin katsotaan lisääntyneen niin paljon, ettei siihen tarvitse enää kiinnittää huomiota.

Yksinkertaisimmat ohjelmat, kuten SourceJammer, tallensivat tiedostot kokonaisina. Myös Vesta toimi samalla tavalla, mitä perusteltiin helpolla pääsyllä aiempiin versioihin. Yksinkertaisuuden lisäksi tapaa perusteltiin myös käyttäjämukavuudella, jonka takaamiseksi esimerkiksi Archin kehittäjä Tom Lordin mielestä kymmenkertainen tilankulutuksen kasvu ei ole merkittävä haitta. Modernit ohjelmat, kuten GIT ja Mercurial, tallensivat tiedot kuitenkin edelleen deltoina.

Yksi syy deltojen käyttöön tallennustilan riittävyydestä huolimatta on hajautettujen versionhallintaohjelmien vaatima tiedonsiirtokapasiteetti. Koska tietoverkkojen nopeudet eivät ole kasvaneet läheskään niin nopeasti kuin tallennustilan määrä,

siirrettävän datan määrä on edelleenkin pidettävä kohtuullisena. Koska tähän tarkoitukseen käytetään joka tapauksessa deltoja, niiden käyttö myös tallennukseen on pieni vaiva.

Käytetyt deltat vaihtelivat ohjelman mukaan. Perinteisten etenevien, takautuvien ja limittäisten deltojen lisäksi käytössä oli myös hyppydeltoja (4.10.3, s. 93) ja NST-deltoja (Nonlinear Suffix Tree Delta) (4.24.3, s. 141). Joissakin tapauksissa käytettiin perinteisten deltojen parannettuja versioita, joissa muutoksia ei rajoituta laskemaan vain peräkkäisistä versioista.

Käytettyyn tallennusmenetelmään oli kiinnitetty huomiota erityisesti Mercurialissa, joka käytti siihen omaa revlog-menetelmää (4.26.1, s. 157). Samantapaista ideaa oli hyödynnetty myös Bazaarin knit-muodossa, joka mahdollisti perinteisen weaven atomisen käytön.

GITissä tilankäyttöön ei kiinnitetty alussa lainkaan huomiota, joten se hävisi sen suhteen selvästi etenkin Mercurialille. Matt Mackallin arvioiden[3] mukaan Linuxytimen kehityshistoria vei GITissä jopa 3,5 gigatavua, kun se Mercurialissa mahtui 297 megatavuun. Arvio oli hieman virheellinen, sillä GITin suhteen oikea luku oli "vain" 2,5 gigatavua. Pakatun historian käyttöönoton myötä tilanne kuitenkin muuttui oleellisesti, kun historian vaatima tila GITissä tippui ennätysalhaiseen 227 megatavuun[4].

Artikkelissaan "Version Models for Software Configuration Management" [9] Conradi ja Westfechtel huomauttavat, että uudemmissa sovelluksissa ollaan siirtymässä entistä enemmän tietokantojen käyttöön. Esimerkkinä tästä mainitaan Adele, jossa siirryttiin tiedostojärjestelmäpohjaisesta tallennusmuodosta oliotietokannan käyttöön. Avoimen lähdekoodin puolelta on kuitenkin löydettävissä esimerkkejä toisensuuntaisesta siirtymästä, sillä esimerkiksi Subversionissa pyritään pikemminkin luopumaan tietokantariippuvuudesta ja siirtymään kokonaan virtuaalisen tiedostojärjestelmän käyttöön.

Näiden erojen syitä voidaan selittää ainakin niin, että joko kaupalliset tietokannat ovat selvästi parempia kuin avoimen lähdekoodin vastaavat, tai sitten avoimen lähdekoodin tiedostojärjestelmät ovat parempia kuin kaupalliset. MySQL:n[5] menestys ei kuitenkaan tue ensimmäistä oletusta, eikä jälkimmäiseen vaadittavaa erottelua kaupallisten ja avoimen lähdekoodin tuotteiden välillä voida enää varmasti tehdä, etenkään kun Sun Microsystems ja Oracle ovat mukana omilla ZFS- ja BTRFS-

[3] http://www.selenic.com/pipermail/mercurial/2005-May/000334.html (viitattu 21.10.2007).

[4] http://www.gelato.unsw.edu.au/archives/git/0505/3947.html (viitattu 21.10.2007).

[5] http://www.mysql.com/ (viitattu 12.11.2007).

tiedostojärjestelmillään[6].

5.3.4 Yhdistysmenetelmä

Yksi osa-alue, mihin ainakin uusimmissa sovelluksissa oli selvästi panostettu, on muutosten yhdistäminen. Tämä ei ollut kuitenkaan mikään yllätys, sillä avoimen lähdekoodin projekteihin yleensä liittyvässä hajautetussa kehitysmallissa alkuperäinen lukitse-muokkaa-vapauta-malli (2.9.1, s. 20) muodostuisi nopeasti kaikkia hidastavaksi pullonkaulaksi. Yleisemmin käytettävä kopioi-muokkaa-yhdistä-malli (2.9.2, s. 21) toimiikin tässä tilanteessa paljon paremmin, joten mahdolliseksi ongelmakohdaksi jää ainoastaan tehtyjen muutosten yhdistäminen.

Alkuaikojen yhden käyttäjän versionhallintaohjelmissa, kuten SCCS:ssä ja RCS:ssä, kehityshaarojen yhdistämiseen riitti perinteinen 3-way-merge-algoritmi (2.13.2, s. 35). Uudemmissa monen käyttäjän hajautetuissa järjestelmissä sen ei kuitenkaan katsota enää vastaavan tarkoitustaan, joten sitä on paranneltu monin tavoin tai se on korvattu kokonaan jollakin toisella menetelmällä.

Yksi hajautettuun kehitykseen liittyvä ongelma liittyi eri tietovarastojen synkronointiin. Jos *A*:han yhdistetään *B*:ssä tehdyt muutokset, *A*:han tulee siitä merkintä. Tämän jälkeen *A*:n sisältö ei vastaa *B*:tä, joka pitää vuorostaan päivittää. *B*:n päivityksen jälkeen se ei vastaa *A*:ta, joka pitää jälleen päivittää… GNU Archin ratkaisu tähän loppumattomaan päivityskierteeseen oli star-merge (4.12.2, s. 101), joka osaa rajoittua oleellisten muutosten yhdistämiseen.

Codevillessä pyrittiin puolestaan ratkomaan tilanteita, joissa 3-way-merge toimii väärin. Eräs tällainen esimerkkitilanne on esitetty kohdassa 4.16.3, ja siitä selviävä *Simple Weave Merge* -algoritmi kohdassa 4.16.4. Kyseistä algoritmia oli käytetty myös Bazaarin *Knit Merge*n (4.22.3, s. 135) lähtökohtana.

GITissä käytettiin omaa rekursiivista 3-way-merge-algoritmia (4.25.7, s. 150), joka osasi ratkaista yksinkertaisimmat ristiriitatilanteet. Myös monimutkaisten konfliktien automaattinen selvittely oli mahdollista, sillä käyttäjä voi halutessaan “opettaa” järjestelmälle jonkin tilanteen ratkaisun nauhoittamalla oman ratkaisumallinsa komennolla `git rerere`. Jos mallin mukainen ristiriita myöhemmin tulee esiin, järjestelmä osaa selvittää sen automaattisesti.

Bkyssä ei ollut lainkaan omaa yhdistysmenetelmää, joten ristiriitojen selvittely jäi siinä kokonaan käyttäjän vastuulle. Tämä puute ei ollut kuitenkaan mitään ver-

[6] http://www.opensolaris.org/os/community/zfs/, http://oss.oracle.com/projects/btrfs/ (viitattu 12.11.2007).

rattuna FastCST:n nykyiseen yhdistysmenetelmään, jossa käyttäjän tulee etukäteen huolehtia siitä, ettei konflikteja pääse syntymään.

Artikkelissaan "State-of-the-Art Survey on Software Merging" [44] Tom Mens toteaa kaupallisten sovellusten käyttävän lähes poikkeuksetta ainoastaan tekstipohjaisia yhdistysmenetelmiä, mikä hänen mukaansa tekee niistä sopimattomia joihinkin tilanteisiin. Tilanne on kuitenkin sama myös tutkittujen sovellusten osalta, joten voidaan kysyä, onko esimerkiksi semanttisen tai operaatiopohjaisen yhdistämisen tarve vielä puhtaasti teoreettinen, vai onko sille olemassa jo todellista käyttöä.

Toinen Mensin peräänkuuluttama ominaisuus on useamman kuin kahden eri kehityshaaran samanaikainen yhdistäminen. Tätä tarvetta hän perustelee viittaamalla Perryn ym. tutkimukseen [52], jossa n. 45 prosentissa tiedostoista havaittiin olevan samanaikaisesti kehityksessä 2–16 rinnakkaista versiota. Silloin käytössä olleet menetelmät eivät mahdollistaneet kuin kahden eri version yhdistämisen, mutta nykyisin tilanne on ainakin avoimen lähdekoodin puolella muuttunut. GITin "octopus merge" -algoritmilla on todellisessa tilanteessa yhdistetty 12 eri kehityshaaraa[7], joten ainakin tältä osin Mensin toive on toteutunut.

Yksi mielenkiintoisimmista yhdistämismenetelmiin littyvistä havainnoista tehtiin Vestan (4.8, s. 86) yhteydessä. Tästä suuryrityksen kehittämästä ja käyttämästä järjestelmästä puuttui kokonaan oma tuki yhdistämiselle, johon tarkoitukseen kehotettiin käyttämään ulkopuolista skriptiä. Erityisen mielenkiintoiseksi asian tekee kuitenkin se, että nyt Vestaan ollaan lisäämässä omaa yhdistämistukea, joka tulee suunnitelmien mukaan perustumaan Precise Codeville Merge -algoritmiin. Tämä osoittaa sen, että avoimen lähdekoodin puolella saadaan joskus aikaan sellaisia keksintöjä, jotka kelpaavat myös teollisuuden (tässä tapauksessa Intelin) käyttöön.

5.3.5 Tietovarasto

Erilaisia tietovarastotyyppejä on aiemmin käsitelty kohdassa 2.7 sivulta 17 alkaen. Niiden kehittyminen on ollut asteittaista siten, että paikallisista tietovarastoista on verkottumisen seurauksena siirrytty ensin keskitettyihin ja myöhemmin hajautettuihin malleihin. Tämä kehitys on selvästi havaittavissa eri aikakausien sovelluksissa, joista vanhimmat eivät käytä edes paikallista tietovarastoa, vaan tallentavat käsittelemiensä yksittäisten tiedostojen historian rinnakkaiseen tiedostoon.

Joidenkin sovellusten ainoana tarkoituksena oli lisätä olemassa olevaan ohjel-

[7] http://www.gelato.unsw.edu.au/archives/git/0602/15875.html (viitattu 12.11.2007).

maan tuki uudemman tyyppiselle tietovarastolle. Esimerkiksi DCVS mahdollisti keskitetyn CVS:n käytön hajautettuun kehitykseen, ja SVK lisäsi saman tuen Subversioniin. Aegikseen hajautustuki lisättiin vuonna 1999, ja joitakin poikkeuksia lukuunottamatta tätä uudemmat sovellukset oli suunniteltu alusta asti hajautetuiksi.

Eräs mielenkiintoinen havainto oli verkkolähteissä, jopa Wikipediassa, vallinnut yksimielisyys siitä, ettei Vesta ole hajautettu. Tämä käsitys on kuitenkin virheellinen, ja johtuu Vesta-FAQ:n mukaan siitä, että termit "repository server" ja "repository client" on ymmärretty väärin[8]. Vestaa pidetään keskitettynä lähinnä siksi, että sen käyttö vaatii erillisen palvelimen. Tässä kuitenkin unohdetaan se, että jokaisella käyttäjällä voi olla omalla tietokoneella käynnissä oma palvelin, jolta toiset voivat hakea tietoja. Jokainen tietovarasto on siis itsenäinen kokonaisuus, joten Vestakin on hajautettu järjestelmä.

5.3.6 Käyttöliittymä ja -ympäristö

Eräs merkittävästi ohjelman suosioon vaikuttava tekijä on sen tukema käyttöympäristö. Mikäli sovellus ei toimi potentiaalisen käyttäjäehdokkaan järjestelmässä, vakuuttavakaan myyntipuhe ei auta. Sovelluksen toimimattomuuteen voi olla useita syitä, joista seuraava voidaan laajentaa koskemaan laitteiston lisäksi myös käyttöympäristöä [40]:

> [...] machines (hardware) influence a programmer's decision about which type of software to be produced critically.

Suurin osa tutkituista ohjelmista toimi Unix-järjestelmissä. Osa ohjelmista käytti Java-kieltä, joten ne olivat periaatteessa järjestelmäriippumattomia. Tämän lisäksi muutama muukin ohjelma oli tehty siirrettävillä kielillä.

Tutkimuksessa havaittu Unix-järjestelmien hyvä tuki on mahdollista selittää tutkimusaineiston valinnalla. Koska ainoastaan Windows-järjestelmässä toimivia ohjelmia ei voitu testata käytännössä, tutkimuskohteeksi valikoitui empiiriseen havainnointiin sopivia Linuxissa toimivia sovelluksia. Toisena syynä saattoi olla se, että avoimen lähdekoodin suosijat käyttävät yleensä avoimia järjestelmiä, joihin Windows ei lukeudu. Ben Collins-Sussman on samaa mieltä[9]:

> Show me any open source project where the core developer group are all "win32 enthusiasts". It doesn't exist.

[8] http://wiki.vestasys.org/VestaFAQ/GeneralQuestions/ (viitattu 2.11.2007).
[9] http://www.advogato.org/article/786.html (viitattu 22.10.2007).

Ohjelmien enemmistö käytti tekstipohjaista käyttöliittymää, mikä viittaa siihen, ettei Windows-käyttäjiä oltu huomioitu. On tietysti mahdollista, että joihinkin näistä on saatavilla graafisia käyttöliittymiä. Myös suosittujen ohjelmistokehitysympäristöjen (engl. IDE) tarjoama käyttöliittymä voi edesauttaa Windows-käyttöä.

Tekstipohjaisten käyttöliittymien suosio johtui myös aiheen rajauksesta. Windows-kehityksessä käytetään yleisimmin kokonaisia ohjelmistokehitysympäristöjä ja niiden omia versionhallintatoimintoja. Unix-järjestelmissä puolestaan käytetään tekstipohjaisia sovelluksia joko suoraan tai erillisen kehitysympäristön osana. Hyvänä esimerkkinä voidaan ottaa GNU Emacs, joka tarjoaa yhtenäisen käyttöliittymän useille versionhallintaohjelmille. Graafisten versionhallintaohjelmien etu tekstipohjaisiin ohjelmiin nähden katoaa, kun versionhallinnalta vaaditaan integroitumista olemassa olevaan järjestelmään.

5.3.7 Atomiset muutokset

Atomisilla muutoksilla tarkoitetaan nimensä mukaisesti muutosten tai oikeammin muutosjoukkojen sellaista käsittelyä, joka onnistuu joko kokonaan tai ei ollenkaan. Ellei tällaista menettelyä tueta, jokin muutos voi jäädä vaillinaiseksi ja aiheuttaa myöhemmin ongelmia.

Tutkituissa sovelluksissa muutosten atomiseen tekoon käytettiin kahta erilaista lähestymistapaa. Ensimmäisessä pyrittiin tietokannoista tuttujen transaktioiden käyttöön, mikä hoidetaan käytännössä kirjaamalla osittain tehdyt muutokset ylös. Jos myöhemmin lokitiedostosta huomataan, ettei kaikkia muutoksia ehditty käsitellä ennen keskeytystä, alkutilanteeseen voidaan palata perumalla lokista löytyvät muutokset.

Toinen käsittelytapa perustui tiedostojärjestelmän ominaisuuksien tuntemiseen. POSIX-standardin [34] mukaan tiedostojärjestelmän *rename*-funktion on oltava atominen, joten joissakin järjestelmissä kaikki muutokset tehdään väliaikaiseen tiedostoon, joka siirretään atomisesti vanhan päälle. Mikäli operaatio keskeytyy ennen siirtoa, se voidaan perua yksinkertaisesti poistamalla tarpeettomaksi jäänyt väliaikaistiedosto.

Tutkituista ohjelmista vanhimmat eivät olleet atomisia, mikä on täysin ymmärrettävää. CVS ja muut lukituksia käyttävät ohjelmat pyrkivät tallentamaan kaikki muutokset samalla kertaa, mutta tiedostot tallennettiin ei-atomisesti. Uusimmissa ohjelmissa käytettiin vaihtelevasti kumpaakin yllä mainittua lähestymistapaa.

Eräs atomisuuteen vaikuttava seikka oli se, käyttikö ohjelma ulkoisia apuohjel-

mia. Esimerkiksi yleisesti käytössä oleva *GNU patch* ei ole atominen, eli virhe saattaa johtaa puolittain tehtyyn muutokseen. Tätä pyrittiin korjaamaan mm. tarkistamalla muutostiedostojen oikeellisuus ennen niiden käyttöä, mutta sekään ei auta vaikkapa sähkökatkoksen keskeyttämiin päivitystilanteisiin.

Minkään ohjelman käyttämä keino ei ollut kovin innovatiivinen, mikä selittyy jo olemassa olevien mekanismien toimivuudella. Nykyaikaisten tiedostojärjestelmien ansiosta versionhallintaohjelmat voivat jättää atomisuusvaatimukset käyttöjärjestelmän huoleksi.

5.3.8 Uudelleennimeäminen

Versionhallinnan alkuaikojen sovelluksissa ei ollut tarvetta tukea erillistä uudelleennimeämistä, sillä ne käsittelivät yksittäisiä tiedostoja. Mikäli tiedoston nimen halusi vaihtaa, se onnistui helposti muuttamalla datatiedoston nimi toiseksi. Esimerkiksi RCS:n tapauksessa nimi *vanha,v* voitiin muuttaa suoraan muotoon *uusi,v*, eikä se vaikuttanut aikaisempaan kehityshistoriaan millään tavalla. Uudemmissa kokonaisia tiedostojoukkoja käsittelevissä sovelluksissa tilanne on kuitenkin toinen, sillä niissä nimen vaihto voi vaikuttaa koko projektin tilaan, joten siihen on kiinnitettävä entistä enemmän huomiota.

Kuten aiemmin esitettiin (2.14, s. 38), tiedostojen siirrot ja niiden nimien vaihdot voidaan hoitaa joko eksplisiittisesti tai implisiittisesti. Monissa verkkolähteissä jälkimmäistä tapaa ei kuitenkaan pidetä “oikeana”, joten niissä esimerkiksi väitetään, ettei GIT tue tiedostojen uudelleennimeämisiä. Tämä asenne on helposti ymmärrettävissä, sillä implisiittinen tapa on vielä niin vähän käytetty, ettei sen toimivuudesta ole saatu riittävästi näyttöä.

Tarkastelluista sovelluksista GIT oli ainoa, missä uudelleennimeämiset hoidettiin vain implisiittisesti, sillä kaikissa muissa uudelleennimeämistä tukevissa sovelluksissa käytettiin joko molempia tai vain eksplisiittistä tapaa. Joissakin tapauksissa, esimerkiksi Meta-CVS:n *grab*-toiminnossa, heuristiikkoihin perustuvaa tunnistustapaa voitiin kuitenkin käyttää versionhallinnan ulkopuolella tehtyjen muutosten selvittelyyn.

5.3.9 Historian säilyttävä kopiointi

Tiedostojen ja hakemistojen kopioita voidaan käsitellä joko täysin uusina objekteina, joilla ei ole aiempaa historiaa, tai niihin voidaan liittää alkuperäinen kehityshistoria.

Jälkimmäinen tapa on yleisimmin käytössä niissä sovelluksissa, joissa uudelleennimeämiset käsitellään kopiointeina ja poistoina.

Versionhallinnan alkuaikojen SCCS- ja RCS-järjestelmissä ei ollut erityistä tukea tiedostojen kopioinnille, mutta ne onnistuivat samaan tapaan kuin aiemmin selostettu *,v*-tiedoston uudelleennimeäminen. Monissa uudemmissa sovelluksissa historian säilyttävä kopiointi tuli eräänlaisena uudelleennimeämisen sivutuotteena, sillä sitä ei oltu toteutettu siitä erikseen missään ohjelmassa.

Suhtautuminen tähän toimintoon vaihteli eri projektien välillä. Erimielisyyttä oli mm. siitä, pitäisikö versionhallinnan ulkopuolelta tullutta tiedostoa käsitellä täysin uutena, vai voiko sen merkitä alkaneeksi jostakin sopivasta historian kohdasta. Jälkimmäisestä aiheutuu ongelmia ainakin niissä järjestelmissä, joissa tiedostojen tunnisteet lasketaan niiden sisällöstä ja sijainti versiograafissa sekä sisällön että edellisten versioiden perusteella: jos sekä sisältö että historia ovat samat, kahta tiedostoa ei voi erottaa toisistaan.

Tilapohjaiset (engl. snapshot based) ohjelmat eivät seuranneet yksittäisten tiedostojen historiaa, joten ne joutuivat selvittämään sen käymällä läpi kaikki muutokset. Se, mikä katsottiin kopioidun tiedoston historiaksi, riippui täysin toteutustavasta.

5.3.10 Rivikohtainen historia

Monissa sovelluksissa on mahdollisuus selvittää, mikä muutos on viimeksi vaikuttanut johonkin tiettyyn riviin, ja kuka sen on tehnyt. Tämä toiminto on varsin usein käytössä niissä ohjelmissa, joissa historia tallennetaan tiedostokohtaisesti. Weave-pohjaista tallennusmuotoa käytettäessä muutokset tallennetaan suoraan rivien tarkkuudella, mutta tilannekuvia käyttävissä järjestelmissä tämän *annotate*- tai *blame*-toiminnon toteutus on hieman raskaampi.

Alkuperäinen tapa selvittää rivikohtainen historia perustui CVS:n *annotate*-toimintoon. Myöhemmin toiminnallisuus on toteutettu moniin ohjelmiin joko sisäisesti tai erillisen apuohjelman avulla. Joissakin sovelluksissa toiminnan nimenä on *blame*, mikä viittaa mahdollisuuteen löytää syyllinen rivillä olevaan virheeseen. Subversionissa asian voi nähdä myös toisin, joten käytössä on synonyymi *praise*, jonka avulla tunnustus hyvästä työstä osataan antaa oikealle henkilölle.

Tiedostokohtaiset muutokset tallentavissa järjestelmissä toiminto on yleensä nopea. Muutosjoukkopohjaiset sovellukset joutuvat tekemään enemmän työtä, koska niiden on selvitettävä jokaisesta muutoksesta niiden vaikutukset tutkittaviin rivei-

hin. Tiedostojen uudelleennimeäminen tuo myös omat ongelmansa.

Tutkituista ohjelmista kaikki CVS:ään perustuvat järjestelmät tukivat rivikohtaisen historian näyttöä suoraan. Näiden lisäksi uudemmat ohjelmat tukivat toimintoa suoraan, ja joidenkin ohjelmien kanssa pystyi käyttämään erillisiä apuohjelmia. Esimerkiksi `blame`[10] tarjosi toiminnon RCS:n käyttäjille. Varsinaisia innovaatioita rivikohtaisen historian näytössä ei ohjelmista havaittu.

5.3.11 Historian eheyden varmistus

Hajautetussa kehityksessä, missä tietoja siirrellään useiden eri tietovarastojen välillä, on erittäin tärkeää, että käsiteltävien tietojen eheys voidaan jotenkin varmistaa. Ellei näin tehdä, jokin harmittomaksi tiedetty muutos voi väärästä paikasta ladattuna osoittautua joksikin aivan muuksi ja muodostaa vakavan tietoturvaongelman.

Yksi usein käytetty tapa historian eheyden varmistamiseen on laskea kaikista muutoksista kryptografisesti vahva tarkistussumma eli tunniste[11], jolla niiden sisällön eheys voidaan varmistaa. Jälkeenpäin tehdyt muutokset havaitaan siitä, ettei uudestaan laskettava tunniste enää täsmää alkuperäisen kanssa. Tarkastelluista sovelluksista useimmat käyttävät tähän tarkoitukseen SHA-1-algoritmia (Secure Hash Algorithm), joka tuottaa 160-bittisen tunnisteen.

Koska erilaista dataa on periaatteessa äärettömästi, mutta tunnisteet ovat kiinteämittaisia, kaikelle datalle ei voi antaa yksilöllistä tunnistetta. Tilannetta, missä kahdesta eri tiedostosta saadaan sama tunniste, kutsutaan *yhteentörmäykseksi* (engl. collision). Tällaista tilannetta pidetään kuitenkin niin epätodennäköisenä, ettei siihen aina edes kiinnitetä huomiota. Monotone-listalla ongelman merkityksettömyys esitetäänkin näin[12]:

> So, if you're a billion billion billion times more worried about a hash collision than about your whole family dying in a car collision, then maybe monotone isn't for you.

Kaikki eivät kuitenkaan ole aivan näin vakuuttuneita, joten esimerkiksi ArXissa käyttöön on otettu SHA-1:tä turvallisempana pidetty SHA-256-algoritmi.

[10] http://blame.sourceforge.net/ (viitattu 9.11.2007).

[11] Usein käytetään myös termiä *tiiviste*, joka kuvaa hyvin sitä, että saatu tunniste periaatteessa sisältää tiivistetyssä muodossa kaiken oleellisen alkuperäisestä datasta. Ainakaan vielä näitä tiivisteitä ei kuitenkaan osata purkaa takaisin alkuperäiseksi dataksi, joten niitä ei voi käyttää pakkaamiseen.

[12] http://lists.gnu.org/archive/html/monotone-devel/2005-04/msg00234.html (viitattu 1.11.2007).

Tarkasteltujen sovellusten perusteella oli selvää, että luotettavan historian vaatiminen on melko uusi asia. Esimerkiksi SCCS:ssä käytettävä tarkistussumma osoittautui niin heikoksi, että mahdollinen tunkeutuja voi muutosten teon jälkeen päivittää sen itse, jolloin muutokset jäävät havaitsematta. Tätäkin huonompi tilanne oli RCS:ssä, missä ei ollut mitään tarkistuksia. Vestan tilanne jäi vielä avoimeksi: siinä käytetään sisäisesti omia 128-bittisiä tunnisteita, joiden vahvuudesta ei ole tarkkaa tietoa.

Mielenkiintoinen havainto oli, ettei yhdessäkään kryptografisesti vahvaa algoritmia käyttävässä sovelluksessa (paitsi ehkä Vestassa) käytetty mitään omatekoista tarkistusalgoritmia, vaan kaikissa pitäydyttiin yleisesti tunnetuissa ja toimiviksi tiedetyissä menetelmissä. Yhtenä syynä tähän oli tietenkin se, että hyvän kryptografisen algoritmin laatiminen on äärimmäisen vaikeaa ja vaatii sellaista erityisosaamista, mitä avoimen lähdekoodin projekteista ei näyttäisi löytyvän.

5.3.12 Avainsanojen laajentaminen

Avainsanojen laajentaminen tarkoittaa SCCS:n esittelemää tapaa, jossa versionhallintaohjelma korvaa lähdekoodissa tietyllä tavalla merkityt kohdat niiden arvoilla. Esimerkiksi RCS voi muuttaa tiedostossa olevan tekstin `$Id$` muotoon `$Id: main.c,v 1.1 2007/11/01 12:42:00 atr Exp $`. Tarkoituksena on mahdollistaa käytetyn version tunnistus.

Moderneissa järjestelmissä avainsanojen laajentamista ei pidetä hyvänä ominaisuutena, sillä se aiheuttaa ylimääräisiä konflikteja yhdistämistilanteissa. Yleinen mielipide näyttää kallistuvan sille kannalle, ettei kyseinen toiminto kuulu versionhallintaohjelman tehtäviin, vaan sovelluksen julkaisuvaiheen toimenpiteisiin.

Tutkituista ohjelmista vanhimmat, kuten RCS ja SCCS, sekä niihin pohjautuvat järjestelmät tukivat avainsanojen laajentamista. Uudemmat ohjelmat jättivät toiminnon toteuttamatta, tai tarjosivat mahdollisuuden kiertoteitä käyttämällä. Turhien konfliktien välttämisen lisäksi toinen yleinen syy tämän toiminnon pois jättämiseen on siinä, että tiedoston sisällön muuttuessa myös siitä laskettava tunniste muuttuu, jolloin tällaisia tunnisteita käyttävät järjestelmät eivät enää toimisi luotettavasti.

5.3.13 Pääsynvalvonta

Pääsynvalvonnan avulla voidaan varmistaa, ettei tietovarastoa voi käyttää ilman tarvittavia oikeuksia. Yleisimmässä tapauksessa tietojen lukua ei estetä mitenkään,

mutta niiden kirjoitusta varten vaaditaan ns. *commit access*, eli oikeus tallentaa muutoksia.

Paikallisten tietovarastojen yhteydessä käytetään yleensä käyttöjärjestelmän omia tiedosto-oikeuksia, joilla tietojen lukuoikeudet annetaan vain tietyn ryhmän jäsenille, ja tietovaraston suora käsittely versionhallintaohjelman ohi estetään. Mikäli käyttöjärjestelmä ja tiedostojärjestelmä sitä tukevat, ACL:iä (Access Control List) käyttämällä voidaan asettaa tätäkin yksityiskohtaisempia rajoituksia ylläpitäjän haluamalla tavalla.

Erillistä palvelinta käytettäessä oikeuksien hallinta on yleensä paikallista monipuolisempaa, mutta vaatii myös enemmän ylläpitoa. Yksinkertaisten rajoitusten lisäksi palvelin voi esimerkiksi estää muutoksen tallennuksen, jos se rikkoo noudatettavia lähdekoodin muotoilusääntöjä. Näiden ja monien muiden rajoitusten tekoon käytetään yleensä *koukkuja* (engl. hook) tai *triggereitä* eli skriptejä, jotka ajetaan eri toimintojen yhteydessä.

Tarkastelluista sovelluksista yhdestäkään ei löytynyt pääsynvalvonnan suhteen erikoisia ratkaisuja. Tämä selittynee kuitenkin sillä, että tietoturva on versionhallinnan ulkopuolellakin niin merkittävä elementti, että siihen liittyvät yleisemmät ratkaisut on jo tehty muualla, joten niitä voidaan hyödyntää suoraan. Samaan tulokseen päätyivät jo vuonna 1999 myös Frühauf ja Zeller [19], jotka totesivat silloin näin: "Access control is widely used; it has never been a SCM research topic." Tältä osin he näyttävätkin olleen oikeassa, eikä versionhallinnan puolelta nytkään liene odotettavissa tästä asiasta mitään uutta.

5.3.14 Aliprojektit

Aliprojekteilla tarkoitetaan mahdollisuutta käsitellä erillisiä projekteja jonkin muun projektin itsenäisenä osana. Yksinkertaisimmassa tapauksessa tähän riittää, että käytettävä sovellus osaa jättää tietovaraston sisällä olevat toiset tietovarastot huomiotta. Paremman tuen tarjoavissa järjestelmissä aliprojektien käsittely integroituu normaalikäyttöön tätä paremmin, joten niihin voidaan tehdä muutoksia normaaliin tapaan.

CVS:ssä aliprojekteja käsiteltiin itsenäisinä moduuleina, jotka piti erikseen hakea palvelimelta. Subversionissa niitä nimitettiin *ulkoisiksi osiksi* (engl. external item), ja ne päivittyivät automaattisesti pääprojektin yhteydessä. Yksi esimerkki tämänlai-

sesta tilanteesta näkyy vaikkapa MPlayer-mediasoittimen[13] yhteydessä, missä soittimen kehityksessä käytettävä tietovarasto sisältää myös erikseen kehitettävät libavutil-, libavcodec-, libavformat- ja libpostproc-kirjastot.

Aegiksessa oli täysi tuki aliprojekteille, sillä niitä käsiteltiin tavallisina kehityshaaroina. ArXissa ja Darcsissa tällaista tukea ei varsinaisesti ollut, mutta ne sallivat sisäkkäisten tietovarastojen käytön, joita myös Bazaar tuki. Bazaarin tuki oli kuitenkin vielä keskeneräinen, joten se oli piilotettu, eikä sitä suositeltu käytettäväksi. Monotonen aliprojektituki oli puolestaan yksisuuntainen, eli aliprojektin päivitys onnistui, mutta siihen ei saanut tehdä suoraan muutoksia.

GITissä aliprojektien käyttöön oli kaksi erilaista tapaa. Ennen kunnollisen tuen valmistumista aliprojekteja käsiteltiin normaalien hakemistojen tapaan siten, että niihin tehdyt muutokset yhdistettiin pääprojektissa niille varattuihin hakemistoihin omalla *subtree*-yhdistysmenetelmällä. Myöhemmin aliprojekteille tuli myös "virallinen" tuki, jossa niitä käsiteltiin omalla `submodule`-komennolla. Oletuksena ne eivät päivity automaattisesti pääprojektin kanssa, mutta yksinkertaisen `git submodule update` -komennon sisältävä skripti on helppo lisätä päivityksen jälkeen ajettavien komentojen (engl. hook) joukkoon[14].

5.3.15 Rivinvaihtojen käsittely

Rivinvaihtojen käsittely viittaa tilanteeseen, jossa versioitavien tekstitiedostojen rivinvaihtomerkit vaihtelevat. Tällainen tilanne syntyy lähinnä silloin, kun samaa tiedostoa muokataan eri käyttöjärjestelmissä. Windows-järjestelmissä tekstirivit päättyvät `\r\n`-tavuihin[15], Unix-järjestelmissä `\n`-tavuun ja Mac OS:ssä `\r`-tavuun. Nämä erot aiheuttavat ongelmia etenkin rivipohjaisten diff-algoritmien kohdalla, koska rivin käsite ei ole kaikkialla sama.

Yhtenä tapana on säilyttää rivinvaihtotieto sellaisenaan, jolloin versionhallintaohjelman ei tarvitse huolehtia koko asiasta. Ongelmia tosin syntyy, kun tekstiä kopioidaan tiedostosta toiseen, mikäli tiedostojen rivinvaihtokonventiot poikkeavat toisistaan. Toinen vaihtoehto on tallentaa tiedostot käyttämällä aina etukäteen määriteltyä tapaa.

Oli hieman outoa havaita, ettei yhteistyö tämän asian suhteen toiminut. Sama ongelma nousi esille useilla kehityslistoilla eri aikoina, mutta jokaisessa tapaukses-

[13] http://www.mplayerhq.hu/ (viitattu 1.11.2007).

[14] http://lkml.org/lkml/2007/5/20/36/ (viitattu 1.11.2007).

[15] `\r` = CR = Carriage Return, `\n` = LF = Line Feed

sa siihen pyrittiin löytämään jokin oma ratkaisu. Koska kaikki näyttävät kuitenkin lopulta päätyneen itsenäisesti samantapaisiin ratkaisuihin, voidaan kysyä, miksei aiemmin toimivaksi osoittautuneita ratkaisuja hyödynnetty suoraan. Ehkä kyseessä on jälleen malliesimerkki avoimen lähdekoodin projekteja piinaavasta NIH-syndroomasta (Not Invented Here)[16], joka pakottaa tekemään kaiken itse. Tieteellisessä kirjallisuudessa asiaan ei puolestaan kiinnitetty mitään huomiota, mikä saattaa johtua siitä, että sitä pidetään vain turhana toteutusyksityiskohtana.

5.3.16 Merkittävät käyttäjät

Oli mielenkiintoista havaita, kuinka eri käyttöjärjestelmien tai niiden ydinten kehitykseen käytettiin eri versionhallintaohjelmia. FreeBSD- ja OpenBSD-projektit käyttivät perinteistä CVS:ää ja ReactOS sen seuraajaa Subversionia. Linux-kehityksessä käytettiin sitä varten toteutettua GITiä, jonka kilpailija Mercurial puolestaan valittiin OpenSolaris-projektin käyttöön.

Eräs sovellusten käyttöön vaikuttava seikka on niiden saatavuus. Nykyään kovinkaan moni käyttäjä ei hanki sovelluksia lähdekoodimuodossa, vaan pitäytyy jonkin valmiin jakelupaketin mukana tulevissa ohjelmissa. Jonkinlaisen käsityksen sovellusten merkittävyydestä voikin saada tutkimalla, mitä niistä on hyväksytty mukaan johonkin tunnettuun jakelupakettiin. Esimerkiksi 13.11.2007 tarkistetussa Fedora-jakelussa[17] olivat mukana Arch (tla), Bazaar (bzr), CVS, Darcs, GIT, Mercurial, Monotone, RCS, Subversion ja SVK (perl-SVK), joten niiden käyttöönotto on selvästi yksinkertaisempaa kuin muiden tarkasteltujen sovellusten.

Joillekin sovelluksille oli mahdotonta löytää käyttäjiä, koska niiden käyttö ei näy ulospäin. Esimerkiksi jokainen Subversionin tietovarasto on periaatteessa myös SVK:n tietovarasto, joten niiden käyttäjiä ei voinut erotella toisistaan. Tilanne oli sama kaikkien niiden jonkin toisen versionhallintaohjelman päällä toimivien sovellusten kanssa, joiden vaikutus rajoittui joihinkin paikallisiin parannuksiin.

5.4 Sovellusten erityisominaisuudet

Edellä esitettyjen useimmille sovelluksille yhteisten ominaisuuksien lisäksi monista ohjelmista löytyi myös sellaisia erityispiirteitä tai ominaisuuksia, jotka puuttuivat

[16] Käsite on selostettu varsin hyvin Wikipediassa, joten sitä on turha selostaa tässä tarkemmin. http://en.wikipedia.org/wiki/Not_Invented_Here (viitattu 9.11.2007).

[17] http://www.fedoraproject.org/

muista ohjelmista.

5.4.1 Muutosten pilkkominen

Hyvän tavan mukaisesti tehdyt muutokset ovat sellaisia, että ne tekevät vain yhden loogisen muutoksen. Isommat muutokset saadaan aikaan tekemällä useita loogisia muutoksia peräkkäin. Tällä tavalla tehtyjä muutoksia on helpompi ymmärtää kuin jos kaikki muutokset olisi tehty kerralla. Myös virheellisten kohtien löytäminen helpottuu, kun jokainen muutos voidaan testata erikseen.

Yleensä ohjelmankehitys ei kuitenkaan etene niin loogisesti kuin edellä esitetty hyvä tapa vaatisi. Usein kehittäjät tekevät pieniä toisiinsa liittymättömiä muutoksia sinne tänne eri lähdekooditiedostoihin, ja tallentavat kaiken yhtenä isona muutoksena. Näin voi käydä mm. silloin, jos joku ohimennen silmäilee koodia ja korjailee löytämiään kirjoitusvirheitä ja tekee samalla muita pieniä muutoksia. Tästä voi koitua ongelmia myöhemmin, jos jokin muutos poistetaan virheellisenä: samalla menetetään kaikki samanaikaisesti tehdyt parannukset, jolloin kerran korjatut kirjoitusvirheetkin palaavat takaisin.

Tällaisten tapausten estämiseksi on kehitetty keinoja pilkkoa tehtyjä muutoksia loogisiksi kokonaisuuksiksi.

Darcsissa pilkkominen tapahtui muutosta tallennettaessa. Interaktiivinen toiminto kävi kaikki muutokset läpi ja kysyi jokaisen kohdalla, otetaanko se mukaan vai ei. Tällöin esimerkiksi kirjoitusvirheiden korjaukset voitiin aluksi ohittaa ja tehdä niistä myöhemmin erillinen muutos. Samanlainen toiminto oli löytänyt tiensä myös GITiin ja Mercurialiin, mutta esimerkiksi tutkituista konfiguraationhallintaohjelmista (Aegis ja Vesta) ei löytynyt mitään vastaavaa; niissä muutokset pidettiin loogisesti yhtenäisenä muiden keinojen, kuten testiskriptien, avulla.

Vaikka pilkkomistoimintoa ei kaikissa ohjelmissa ollutkaan, samanlainen lopputulos voidaan saavuttaa erillisten kehityshaarojen avulla. Tällöin varsinainen ohjelmankehitys tehdään erillisessä haarassa, ja lopulliset muutokset siirretään päähaaraan *diff*- ja *patch*-ohjelmien avulla, mikä mahdollistaa muutosten pilkkomisen ulkoisia apuohjelmia käyttämällä.

Joissakin ohjelmissa, kuten GITissä, muutoksia pystyi pilkkomisen lisäksi myös yhdistelemään ja järjestelemään uudelleen. Toiminnon avulla esimerkiksi bugikorjaukset voitiin liittää suoraan alkuperäisen muutoksen osaksi, mikä johti versiohistorian siistiytymiseen ja huomattavasti helpompaan virheenetsintään.

5.4.2 Virheenetsintä

Ohjelmistonkehityksessä tulee aina virheitä, riippumatta siitä, miten hyvin koodi on tarkastettu ennen sen mukaanottoa. Virheet voivat olla piilossa kauan, ennenkuin jokin muutos aktivoi ne, joten niiden etsintää ei voi rajoittaa vain uusimpiin versioihin. Testitapauksetkaan eivät välttämättä kata kaikkia toimintoja, joten toimivaksi tiedetyt versiot voivat sisältää samoja virheitä.

Virheidenetsinnässä pyritään löytämään ensimmäinen ohjelmaversio, jossa kyseisen virheen aiheuttaja on otettu mukaan. Tämä voidaan tehdä monella eri tavalla, joko käsin tai automaattisesti.

Yksinkertaisin tapa on testata eri versioita takenevassa järjestyksessä siten, että uudemmista versioista siirrytään aikaisempiin poistamalla niihin tehtyjä muutoksia. Tämä voi toimia silloin, kun virhe on suhteellisen uusi, mutta yleiseksi ratkaisuksi tästä ei ole. Lisäongelmia aiheutuu tilanteista, joissa viimeisin testattu versio onkin kahden kehityshaaran yhdistelmä. Tällöin edellisen version valinta ei olekaan aivan selvää.

Ratkaisu tähän ongelmaan löytyi GITistä, jossa *bisect*-toiminto otettiin käyttöön 18.6.2005[18]. Kyseinen toiminto toimi puolitushaun tavoin, eli valitsi testattavia versioita siten, että jokaisen version testauksella jäljelle jäävien testauskohtien määrä puolittui. Vaikka idea ei ollut uusi, sitä ei oltu toteutettu aiemmin tätä tarkoitusta varten.

Bisect-toiminnon käyttö vaatii siistin kehityshistorian, eli jokaisen vanhan version on oltava toimintakunnossa. Tällöin esimerkiksi loogiset muutokset on tehtävä yhtenä kokonaisuutena. Rajoitus voi haitata toiminnon käyttöä sellaisissa projekteissa, joissa olemassa oleva versiohistoria ei täytä tätä ehtoa.

GITin esittelemä toiminto on otettu käyttöön monissa järjestelmissä joko suoraan (esim. Mercurial[19]) tai erillisen apuohjelman avulla (esim. CVS[20]). Toiminto onnistuu myös manuaalisesti, joten se on helppo toteuttaa komentotiedostojen avulla mille tahansa ohjelmalle. Sen toimivuus on osoitettu käytännössä etenkin Linux-kehityksessä, missä virheistä raportoivat käyttäjät ovat alkaneet liittää ilmoituksiinsa myös tiedon siitä, mikä muutos kyseisen virheen on aiheuttanut. Koska tällainen ei onnistuisi suljetun lähdekoodin projekteissa, voidaan olettaa, ettei toimintoa tulla hyödyntämään julkisesti muualla kuin avoimen lähdekoodin projekteissa.

[18] http://www.gelato.unsw.edu.au/archives/git/0506/5401.html

[19] http://www.selenic.com/mercurial/wiki/index.cgi/BisectExtension/ (viitattu 9.11.2007).

[20] http://wingolog.org/archives/2006/01/20/cvs-bisect (viitattu 9.11.2007).

5.4.3 Muutosten levitys

Kun kehittäjä on tehnyt lähdekoodiin muutoksia, ne on saatettava muiden käyttöön. Tämän voi tehdä monella eri tavalla, käytetystä versionhallintaohjelmasta riippuen.

Yleisin tapa on käyttää versionhallintaohjelman omaa toimintoa, jolla muutokset siirretään yhteiseen tietovarastoon. Tämä voi tapahtua myös automaattisesti, mikäli käytetään keskitettyä tietovarastoa, kuten esimerkiksi CVS:n tapauksessa.

Hajautetuissa versionhallintaohjelmissa käytössä on yleensä ns. *push-* ja *pull-*toiminnot, joilla muutokset joko siirretään toiseen tietovarastoon tai haetaan paikalliseen tietovarastoon. DCVS:ssä tähän tarkoitukseen käytettiin replikointiverkkoa (4.13.2), jolla konfliktien esiintymistä voidaan vähentää.

Eräs versionhallintaohjelmasta riippumaton tapa muutosten levittämiseen on perinteisten muutostiedostojen eli ns. *patchien* käyttö. Tällöin muutos siirretään tekstimuotoisena tiedostona esimerkiksi sähköpostin välityksellä, jolloin muutoksen vastaanottaja voi laittaa sen paikoilleen joko käyttöjärjestelmän omien työkalujen tai versionhallintaohjelman tähän tarkoitukseen tarjoamien toimintojen avulla. Joissakin sovelluksissa (mm. GIT ja Mercurial) on myös mahdollisuus ns. *bundle*n käyttöön, jolloin useita muutoksia voidaan lähettää yhteen tiedostoon pakattuna.

Eräs mielenkiintoinen, vielä idea-asteella oleva mahdollisuus muutosten levitykseen löytyi Fonsecalta [18], joka esitteli tavan käyttää siihen vertaisverkkoa. Tätä *GitTorrentiksi*[21] nimitettyä protokollaa käyttävä sovellus oli tarkoitus toteuttaa Googlen Summer of Code -projektina kesän 2007 aikana, mutta se jäi kuitenkin toteutumatta[22]:

> Unfortuantely [sic] this isn't a project under GSoC anymore. The student involved had some issues with being able to get his paperwork completed for Google and was not able to participate this summer. I have not heard much more than that.

Vaikka projekti ei onnistunutkaan, se saattaa tulevaisuudessa johtaa uudenlaisiin vertaisverkkoja hyödyntäviin ratkaisuihin. Tarvittava teknologiahan on jo olemassa, joten sen luova soveltaminen on vain ajan kysymys. On kuitenkin mielenkiintoista, ettei alkuperäisen BitTorrentin ja Codevillen kehittäjä Bram Cohen ole itse aiemmin keksinyt yhdistää näitä kahta projektiaan.

[21] Nimi on tietenkin muunnos alkuperäisestä BitTorrentista.

[22] http://lists.zerezo.com/git/msg624647.html (viitattu 25.8.2007).

GitTorrent on hyvä esimerkki siitä, että olemassa olevien ideoiden luova yhdistely voi avata aivan uusia mahdollisuuksia. Hyvä esimerkki tällaisesta on myös repo.or.cz-sivuston tarjoama *mob*-palvelu[23], joka antaa mahdollisuuden osallistua sovellusten kehittämiseen nimettömänä. Se toimii hieman samaan tapaan kuin Wikipedia, paitsi että muutokset otetaan käyttöön vasta sitten, kun tietovaraston ylläpitäjä on tarkistanut ne. Osoitus tämänkaltaisen kehityksen toimivuudesta on jo saatu esimerkiksi GITin graafisen käyttöliittymän, *git-gui*n, muuttamisessa monikieliseksi[24].

5.4.4 Kehityshistorian siistiminen

Eräs osa-alue, mihin ainakin menestyneimmissä avoimen lähdekoodin projekteissa selvästi panostetaan, on lähdekoodin pitäminen siistinä. Yksi syy tähän on tietenkin se, ettei julkisuuteen haluta laittaa huonolaatuista koodia, joka antaisi projektista huonon kuvan. Tämä on kuitenkin monesti johtanut siihen, että versionhallinnan alle on laitettu vasta lopullinen lähdekoodi, joka on kehitetty versionhallinnan ulkopuolella.

Edellisen kaltaiset ongelmat johtuivat pääasiassa keskitetyn versionhallinnan käytöstä, jossa jokainen tallennettu muutos tuli julkiseksi. Hajautetussa mallissa tätä ongelmaa ei kuitenkaan ole, koska kehittäjä voi itse päättää, mitä ja milloin julkaisee. Tämä uusi vapaus voi kuitenkin aiheuttaa sen, ettei lähdekoodin laatuun kiinnitetä tarpeeksi huomiota heti alussa, joten kehityshistoria on siistittävä jälkeenpäin.

Kehityshistorian siisteyteen kiinnitetään huomiota erityisesti Linux-ytimen kehityksessä, missä jokaisen muutoksen on oltava itsenäinen niin, että lähdekoodi on käyttökelpoinen missä tahansa historian vaiheessa. Kehityksessä käytettävä GIT tarjoaa tätä varten mm. aiemmin esitellyt (s. 151) *rebase*- ja *filter-branch*-toiminnot sekä *cherry-pick*-komennon, jolla historiasta voidaan poimia yksittäisiä muutoksia. Darcsissa tietovaraston sisältöä voidaan puolestaan muuttaa interaktiivisella *record*-toiminnolla.

5.4.5 Päivitysteoria

Darcsin perustana oleva päivitysteoria (4.17.3, s. 122) on yksi niistä avoimen lähdekoodin projekteissa syntyneistä innovaatioista, jotka ovat onnistuneet herättämään

[23] http://repo.or.cz/mob.html (viitattu 12.7.2007).

[24] http://lists-archives.org/git/624709-git-gui-i18n-repo-on-repo-or-cz.html (viitattu 20.8.2007).

myös tieteellistä mielenkiintoa. Vaikka se alkuperäisessä muodossaan olikin tieteellisessä mielessä vielä raakile, Löh ja kumppanit näkivät siinä olevat mahdollisuudet. Heidän mukaansa juuri alkuperäisen teorian epämääräisyydet toimivat inspiraationa heidän omalle tutkimustyölleen [41]:

> However, the theory is fairly sketchy. Some of the definitions are a bit vague, some proofs are missing, and only few properties are stated. In fact, it was the vagueness of the theory of darcs that inspired our work: we wanted to develop a theory that is general enough to describe darcs, in particular, but can serve as a firm foundation for different flavours of version control systems.

Päivitysteoria onkin hyvä esimerkki siitä, että avoimen lähdekoodin idean mukaisesti alkuun pääsemiseksi riittää vain osoittaa, että idea on toteuttamiskelpoinen. Omalta osaltaan tämä myös tukee tutkielman alussa esitettyä väitettä siitä, että avoimen lähdekoodin versionhallintaohjelmista löytyy myös tieteellisesti mielenkiintoisia ja uutta tutkimusta vaativia asioita.

5.5 Sovellusten yhteentoimivuus

Kaiken edellä esitetyn perusteella voidaan todeta, että uuden ohjelmistoprojektin aloittaja on versionhallintaohjelmaa valitessaan vaikeassa tilanteessa. Perustoiminnot onnistuvat nykyään melkein millä tahansa sovelluksella, mutta erikoisominaisuuksissa niiden välillä on selviä painotuseroja. Paras vaihtoehto olisi tietysti se sovellus, joka vastaa parhaiten olemassa oleviin tarpeisiin, mutta entä sitten, jos sen kehitys yhtäkkiä loppuukin?

Usein käy myös niin, ettei versionhallintaohjelmaa voikaan valita vapaasti. Syynä voi olla esimerkiksi se, että yritys on hankkinut jonkin kaupallisen versionhallintaohjelman käyttölisenssin joskus aiemmin. Tällöin yrityksen johtoporras ei näe syytä jättää kallista ohjelmistoa käyttämättä, vaan vaatii sen käyttöä kaikissa projekteissa. Vaikka tämä ei olekaan projektin kannalta kovin hyödyllistä, kehittäjien on parhaansa mukaan yritettävä mukautua asetettuihin raameihin.

Tällaisissa tilanteissa eri versionhallintaohjelmien välinen yhteensopivuus voi tuoda avun. Parhaassa tapauksessa kehittäjät voivat käyttää itse valitsemaansa työkalua, joka osaa toimia suoraan vanhan järjestelmän kanssa. Suurin osa nykyisistä versionhallintaohjelmista toimiikin ainakin kohtuullisen hyvin toisten ohjelmien kanssa.

Tavallisin yhteentoimivuuden muoto on se, että ohjelma osaa käyttää toisen versionhallintaohjelman tietovarastoa sen verran, että kaikki tarpeellinen tieto saadaan sieltä ulos. Tämä toiminto on riittävä silloin, kun pelkkä tietojen saaminen ulos riittää, esimerkiksi versionhallintaohjelmasta toiseen siirryttäessä. Jotkut sovellukset menevät vieläkin pidemmälle tarjoamalla niin hyvän yhteensopivuuden, ettei käyttäjän tarvitse edes tietää, että taustalla on jokin muu versionhallintaohjelma.

Sovellusten välisen yhteentoimivuuden etuna on, ettei käyttäjän tarvitse sitoutua mihinkään tiettyyn ohjelmaan tai toimittajaan. Avoimen lähdekoodin projekteissa tämä on merkittävä etu, mutta kaupallisella puolella tällaista ei yleensä tarjota. Monissa kaupallisissa sovelluksissa tietojen siirto omaan järjestelmään toimii, mutta niiden siirto muuhun järjestelmään on rajoitettua tai sitä ei tueta ollenkaan.

Eräs mielenkiintoinen tilanne esiintyi Bky:n kohdalla. Kyseisen ohjelman mukana oli kaksi muunnosskriptiä, joista toinen osasi hakea CVS-tietovaraston tiedot Bky:n käyttämään muotoon, ja toinen osasi siirtää Bky:n tiedot GITiin. Muiden ohjelmien tarjoamat muunnokset rajoittuivat yleensä muiden ohjelmien tietojen siirtoon ohjelman omaan formaattiin.

Vaikka monet versionhallintaohjelmat tukevat suoraan toisten ohjelmien tietovarastojen käyttöä, tähän tarkoitukseen on tehty myös erillisiä apuohjelmia. Esimerkiksi *Tailor*[25] on sovellus, joka muuntaa tietoja eri sovellusten käyttämien tietorakenteiden välillä. Taulukossa 5.2 käytetyt ilmaisut *luku* ja *kirjoitus* merkitsevät sitä, että Tailor osaa lukea kyseisen ohjelman käyttämää muotoa tai kirjoittaa kyseiseen muotoon.

Toinen, hieman rajoittuneempi, muunnosohjelma on SVNImporter, joka osaa muuttaa toisista ohjelmista Subversion-muotoon (taulukko 5.3)[26].

Muunnosohjelmien kapasiteetti kasvaa hyvin nopeasti, ja uusia sovelluksia tulee aina silloin tällöin. Tutkielman viimeistelyn aikana ilmestyi *MR*[27], joka tukee muunnoksia useiden tutkittujen ohjelmien välillä. Ohjelma näyttää lupaavalta, mutta valitettavasti sitä ei ehditty testata tämän tutkimuksen puitteissa.

[25] http://progetti.arstecnica.it/tailor/ (viitattu 22.11.2007)

[26] http://www.polarion.org/index.php?page=overview&project=svnimporter

[27] http://kitenet.net/~joey/code/mr/ (viitattu 9.11.2007).

Ohjelma	tuki
Arch 1.0	vain luku
ArX	vain kirjoitus
Bazaar	luku ja kirjoitus
Codeville	vain kirjoitus
CVS	luku ja kirjoitus
Darcs	luku ja kirjoitus
GIT	luku ja kirjoitus
Mercurial	luku ja kirjoitus
Monotone	luku ja kirjoitus
Subversion	luku ja kirjoitus

Taulukko 5.2: Tailorin tukemat muunnokset.

Ohjelma
CVS (cvs2svn)
PVCS (pvcs2svn)
VSS (vss2svn)
ClearCase (cc2svn)
MKS (mks2svn)
StarTeam

Taulukko 5.3: SVNImporterin tukemat lähdeformaatit.

5.6 Jatkotutkimusideoita

Tässä tutkielmassa versionhallintaohjelmien postilista-arkistoja tutkittiin vuodesta 2005 alkaen, eli n. kolmen vuoden ajalta. Jatkoa ajatellen olisi mielenkiintoista mennä arkistoissa vieläkin pidemmälle, ehkä aivan alkuun asti, ja selvittää sieltä käsin, kuinka ohjelmien käyttämät ideat ovat muotoutuneet. Tällä tavoin voisi empiirisesti varmistua, onko avoimen lähdekoodin prosessimalli sellainen kuin sen sanotaan olevan. Samalla näkisi, miten suuri merkitys tieteellisen tutkimuksen tuloksilla on ohjelmistonkehityksen missäkin vaiheessa.

Eräs esille noussut kysymys liittyy tieteellisten tutkimustulosten hyödyntämiseen avoimen lähdekoodin projekteissa. Kuten aiemmin esitettiin, yhdessäkään kryptografisesti vahvaa algoritmitmia käyttävässä sovelluksessa ei käytetty mitään omaa ratkaisua, vaan kaikissa hyödynnettiin jotain tunnettua menetelmää. Toisaalta yh-

distämismenetelmistä löytyi paljon omintakeisia ratkaisuja, jotka olivat joko kokonaan uusia tai sisälsivät selviä parannnuksia yleisesti tunnettuihin menetelmiin.

Edellisen perusteella voidaan kysyä, miksi tilanne on esitetyn kaltainen. Ovatko kryptografian saavutukset niin päteviä, ettei niitä voi enää parannella, vai vaatiiko aihe sellaista osaamista, jota ei löydy muualta kuin tieteen piiristä? Entä ovatko yhdistysmenetelmät tieteellisesti vielä niin huonosti tunnettuja, ettei niiden joukosta löydy edes yhtä toimivaa, vai ovatko ne jo niin perusteellisesti käsiteltyjä, että maallikotkin uskaltavat tarttua niihin?

Näiden kysymysten perinpohjainen selvittely vaatisi tässä tutkielmassa käytetystä poikkeavan menetelmän käyttöä, sillä siinä pitäisi ottaa huomioon mm. yksittäisten kehittäjien taustat. Nyt on esimerkiksi mahdotonta sanoa, onko kehittäjien koulutus vaikuttanut joidenkin innovaatioiden syntyyn.

Mielenkiintoinen huomio oli, että joidenkin merkittävien innovaatioiden taustalla oli henkilöitä, jotka tunnetaan jostain aikaisemmasta yhteydestä, esimerkiksi Linus Torvalds (Linux) ja Bram Cohen (BitTorrent). Se, miten tällaisten henkilöiden mukanaolo vaikuttaa ohjelmien käyttäjämääriin ja siten epäsuorasti potentiaalisiin innovaatioihin, vaatisi lisäselvittelyä.

Alun perin tutkielman aiheesta oli tarkoitus tehdä laadullinen tutkimus, mutta tämä katsottiin liian työlääksi ja laajaksi käsillä olevan tutkielman kannalta. Alkuperäisen idean toteuttaminen voisi kuitenkin tuoda asiaan uusia näkökulmia ja täydentää joitakin kohtia, jotka nyt jouduttiin sivuuttamaan pelkällä maininnalla sopivien lähteiden puutteen vuoksi. Esimerkiksi aineistopohjaisen menetelmän (engl. Grounded Theory) avulla aiheesta voisi hyvinkin saada esiin sellaisia perusteltuja näkökohtia, joita tässäkin tutkielmassa olisi kaivattu omien päätelmien tueksi.

5.7 Yhteenveto

Avoimen lähdekoodin versionhallintaohjelmilla on odotusten mukaisesti paljon yhteistä, mutta monissa sovelluksissa on käytössä myös omintakeisia ratkaisuja. Hyväksi havaittuja menetelmiä käytetään lähes poikkeuksetta sellaisenaan, mutta etenkin yhdistämismenetelmiä on muokattu omiin tarpeisiin sopiviksi.

Tutkimusta aloitettaessa tiedettiin, että ohjelmien taso vaihtelisi huomattavasti. Tämä ennakkokäsitys osoittautui oikeaksi, mutta suurin osa ohjelmista toimi vähintäänkin tyydyttävästi. Kaikki eivät tosin soveltuneet kovin raskaaseen käyttöön, mutta perusideat näyttivät oikeilta.

Tutkimuksessa löytyi jonkin verran innovaatioita, mutta ne kasautuivat muutaman ohjelman osalle. Suuri osa liittyi olemassa olevien algoritmien paranteluun, mutta mukana oli myös pari täysin uutta ideaa. Arvelu, ettei käytössä olevia ideoita tuoda erityisesti esille, osoittautui paikkansapitäväksi.

6 Yhteenveto

> *The SCM basic concepts and technologies may have been settled, but substantial work remains to be done. [...] We look forward to the advances that will come from this research, and are proud to be a part of a field with such a rich legacy as SCM.*
>
> — *Jacky Estublier et al. [16]*

Tutkielmassa esiteltiin versionhallintaa sekä tieteen teoreettisesta että avoimen lähdekoodin käytännöllisestä näkökulmasta. Alkupuolella käytiin läpi aiheeseen liittyviä käsitteitä ja algoritmeja sekä osoitettiin, kuinka ne ovat vuosien saatossa historiallisesti kehittyneet. Tämän jälkeen tutustuttiin avoimen lähdekoodin historiaan ja esiteltiin siihen liittyviä näkemyksiä. Empiirisessä osassa perehdyttiin joukkoon avoimen lähdekoodin versionhallintaohjelmia, joista tehtyjä havaintoja analysoitiin myöhemmin luvussa 5.

Tutkielman yhtenä tavoitteena oli nostaa esiin sellaisia ideoita ja asioita, joita ei ole aiemmin käsitelty tieteellisessä kirjallisuudessa. Tällaisia "löytöjä" olivat mm. revlog-tallennusmenetelmä (s. 157), Non-Linear Suffix Tree Deltas -delta-algoritmi (s. 141) ja Simple Weave Merge -yhdistysalgoritmi (s. 117), sekä idea vertaisverkkojen käytöstä muutosten levitykseen (s. 198). Näiden lisäksi sovelluksista löytyi tunnettuihin menetelmiin tehtyjä parannuksia (hyppydeltat, s. 93), uusia soveltamistapoja (rekursiivinen 3-way-merge, s. 150) ja jopa uusia toimintoja (bisect, s. 155), joiden voidaan olettaa siirtyvän vähitellen myös yleiseen käyttöön. Uudeksi löydöksi voidaan laskea myös päivitysteoria (s. 122), josta laadittu ensimmäinen tieteellinen artikkeli näki päivänvalon vasta tämän tutkielman aloituksen jälkeen.

Tutkimuksessa perehdyttiin erilaisiin avoimen lähdekoodin versionhallintaohjelmiin sekä käytännössä että niihin liittyvän oheismateriaalin avulla. Sovellusten innovatiivisuutta arvioitiin vertaamalla niitä alan kirjallisuudesta löydettäviin ratkaisuihin, mutta myös valmiiden ratkaisujen luova yhdistely katsottiin innovatiiviseksi.

Eräs mielenkiintoisista havainnoista oli se, että tieteellisessä kirjallisuudessa vapaiden ohjelmien ja avoimen lähdekoodin välisiä suhteita käsitellään varsin pinnallisesti; lähteinä ovat yleensä vain näiden liikkeiden edustajien julkaisemat kirjoi-

telmat, tai sitten tutkimukset perustuvat ulkopuolisena tehtyihin havaintoihin. Yhdessäkään lähteessä asioita ei tarkasteltu pintaa syvemmältä, eli pohdittu sitä, mitä eroja näissä liikkeissä oikeasti on ja mistä ne johtuvat. Luvun 3 lopussa sivulla 61 asiasta on pyritty tekemään kiistan molempien osapuolten näkemykset huomioon ottava yhteenveto.

Toinen huomionarvoinen havainto oli tieteen ja avoimen lähdekoodin projektien välisen yhteistyön lähes täydellinen puuttuminen. Etenkin monissa tieteellisissä julkaisuissa ideoiden toimivuus osoitetaan varta vasten esittelyä varten laaditulla sovelluksella, vaikka selvästi parempi vaihtoehto olisi parantaa jotakin olemassa olevaa avoimen lähdekoodin ohjelmaa. Yksi syy tähän on tieteessä käytetty katedraalityyppinen toimintatapa, jonka mukaisesti keksintöjä ei haluta julkistaa liian aikaisessa vaiheessa. Voidaan kuitenkin kysyä, veisikö avoimempi toimintamalli parempaan lopputulokseen…

Kaikkien esiteltyjen asioiden lisäksi tutkielmaan on saatu mukaan myös yksi ainutlaatuinen tieto, jota ei ole missään muualla. Mikäli tarkkaavainen lukija kiinnitti huomionsa johdannon lopussa (s. 6) olevaan mainintaan Wikipediasta ja seurasi alaviitteessä annettua linkkiä, hän päätyi sivulle, jossa suomenkielisen Wikipedian alkuajoista kerrotaan näin:

> 21. helmikuuta 2002: suomenkielinen Wikipedia perustettiin www-osoitteeseen http://fi.wikipedia.com käyttäen UseModWikiä. Ensimmäinen sivu, Etusivu, luotiin 12 minuuttia yli keskiyön ohjelmiston aikaa osoitteesta gateway.jypoly.fi. Kyseisen muokkauksen teki Jyväskylän ammattikorkeakoulun IT-instituutin silloisissa tiloissa Viitaniemessä eräs ohjelmointitekniikan opiskelija. Etusivun sisältö oli: “Tämä on suomenkielinen Wikipedia-sivusto.”

Tämä on konkreettinen esimerkki siitä, ettei mikään versionhallintaohjelma ole täydellinen. Vaikka Wikipedia onkin tallennettu jonkin versionhallintaohjelman syövereihin, sekään ei osaa antaa lopullista vastausta siihen, *kumpi* tutkielman tekijöistä tuon todellisuudessa kirjoitti.

7 Lähteet

[1] American National Standards Institute. *ANSI/ISO/IEC 9899-1999: Programming Languages – C*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1999.

[2] American National Standards Institute and Information Technology Industry Council. *American National Standard for information technology: programming language – Common LISP: ANSI X3.226-1994*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1996.

[3] Ulf Asklund and Lars Bendix. Configuration management for open source software. Technical report, Aalborg University Dept. of CS, January 2001.

[4] Brian Berliner. CVS II: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 341–352, Berkeley, CA, 1990. USENIX Association.

[5] Jim Buffenbarger and Kirk Gruell. A Branching/Merging Strategy for Parallel Software Development. In *SCM*, pages 86–99, 1999.

[6] Randal C. Burns and Darrell D. E. Long. A linear time, constant space differencing algorithm, 1997.

[7] Benjie Chen. *A Serverless, Wide-Area Version Control System*. PhD thesis, Massachusetts Institute of Technology, 2004.

[8] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly, 2004.

[9] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, 1998.

[10] Reidar Conradi and Bernhard Westfechtel. SCM: Status and Future Challenges. In *System Configuration Management*, pages 228–231, 1999.

[11] B. Cornell, P. Dinda, and F. Bustamante. Wayback: A user-level versioning file system for linux. In *Proceedings of USENIX 2004*, 2004.

[12] Susan Dart. Concepts in Configuration Management Systems. In Peter H. Feiler, editor, *Proceedings of the 3rd International Workshop on Software Configuration Management*, pages 1–18, Trondheim, Norway, 1991.

[13] Chris Dibona, Sam Ockman, and Mark Stone, editors. *Open Sources: Voices from the Open Source Revolution*. O'Reilly, February 1999.

[14] M. Elliott and W. Scacchi. Mobilization of software developers: The free software movement, 2004.

[15] Jacky Estublier. Software configuration management: a roadmap. In *ICSE – Future of SE Track*, pages 279–289, 2000.

[16] Jacky Estublier, David B. Leblang, André van der Hoek, Reidar Conradi, Geoffrey Clemm, Walter F. Tichy, and Darcy Wiborg Weber. Impact of software engineering research on the practice of software configuration management. *ACM Trans. Softw. Eng. Methodol.*, 14(4):383–430, 2005.

[17] Karl Fogel and Moshe Bar. *Open Source Development with CVS*. Paraglyph Press, 2003.

[18] Jonas Fonseca. GitTorrent: a P2P-based Storage Backend for git. Technical report, Department of Computer Science, University of Copenhagen, November 2006.

[19] Karol Frühauf and Andreas Zeller. Software Configuration Management: State of the Art, State of the Practice. *Lecture Notes in Computer Science*, 1675:217–227, September 1999.

[20] Cristina Gacek and Budi Arief. The many meanings of open source. *IEEE Software*, 21(1):34–40, 2004.

[21] James Gosling. A redisplay algorithm. In *Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation*, pages 123–129, New York, NY, USA, 1981. ACM Press.

[22] K. A. Grant. Tacit Knowledge Revisited – We Can Still Learn from Polanyi. *The Electronic Journal of Knowledge Management*, 5(2):173–180, 2007.

[23] Dick Grune. Concurrent versions system, a method for independent cooperation. http://www.cs.vu.nl/∼dick/CVS.html, 1986.

[24] Junio C. Hamano. GIT – A Stupid Content Tracker. In *Proceedings of the Linux Symposium, Volume One*, pages 385–394, Ottawa, Ontario, Canada, 2006.

[25] Richard Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160, April 1950.

[26] Øyvind Hauge and Andreas Røsdal. A Survey of Industrial Involvement in Open Source. Master's thesis, Norwegian University of Science and Technology, Department of Computer and Information Science, 2006.

[27] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. The Vesta approach to software configuration management. Technical Report 168, Compaq Systems Research Center, 2001.

[28] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. The Vesta software configuration management system. Technical Report 177, Compaq Systems Research Center, 2002.

[29] Graydon Hoare, Nathaniel Smith, Derek Scherger, Daniel Carosone, and Jeronimo Pellegrini. *Monotone*, 2003. http://venge.net/monotone/monotone.pdf, viitattu 23.10.2006.

[30] James Howison. Taking research to floss-curious engineers and managers. In *FLOSS '07: Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*, page 6, Washington, DC, USA, 2007. IEEE Computer Society.

[31] T. Hung and P.F. Kunz. UNIX code management and distribution. Technical Report SLAC-PUB-5923, Stanford Linear Accelerator Center, Stanford University, Stanford CA 94309, U.S.A., September 1992.

[32] J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Technical Report #41, Computing Science, Bell Laboratories, July 1976.

[33] James J. Hunt and Walter F. Tichy. Extensible Language-Aware Merging. In *ICSM*, pages 511–520, 2002.

[34] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology – Portable Operating System Interface (POSIX) Base Definitions, Issue 6*. IEEE, New York, NY, USA, 2001. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992) Open Group Technical Standard Base Specifications, Issue 6.

[35] Kim Johnson. A descriptive process model for open-source software development. Master's thesis, Department of Computer Science, University of Calgary, 2001.

[36] Simon Peyton Jones and John Hughes (editors). Haskell 98: A non-strict, purely functional language. Technical report, February 1999.

[37] Thomas Keller. Open Source Version Control. Bachelor Thesis, Leipzig University of Applied Sciences, 2006.

[38] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, February 1966.

[39] Steven Levy. *Hackers: heroes of the computer revolution*. Doubleday, New York, NY, USA, 1984.

[40] Yuwei Lin. *Hacking Practices and Software Development: A Social Worlds Analysis of ICT Innovation and the Role of Open Source Software*. PhD thesis, University of York, September 2004.

[41] Andres Löh, Wouter Swierstra, and Daan Leijen. A Principled Approach to Version Control. http://people.cs.uu.nl/andres/VersionControl.html (viitattu 9.8.2007), 2006.

[42] Josh MacDonald, Paul N. Hilfinger, and Luigi Semenzato. PRCS: The Project Revision Control System. In *System Configuration Management, ECOOP'98 SCM-8 Symposium, Brussels, Belgium, July 20–21, 1998, Proceedings*, pages 33–45, 1998.

[43] Matt Mackall. Towards a better SCM: Revlog and Mercurial. In *Proceedings of the Linux Symposium, Volume Two*, pages 91–98, Ottawa, Ontario, Canada, 2006.

[44] Tom Mens. A State-of-the-Art Survey on Software Merging. *IEEE Trans. Software Eng.*, 28(5):449–462, 2002.

[45] A. Midha. Software configuration management for the 21st century, 1997.

[46] Peter Miller. Aegis — A Project Change Supervisor, 1993. http://aegis.sourceforge.net/auug93.pdf.

[47] Eugene W. Myers. An O(ND) Difference Algorithm and Its Variations. *Algorithmica*, 1(2):251–266, 1986.

[48] Christopher Oezbek and Lutz Prechelt. On Understanding How to Introduce an Innovation to an Open Source Project. In *Proceedings of the 29th International Conference on Software Engineering Workshops (ICSEW '07)*. IEEE Computer Society, IEEE, May 2007.

[49] Bryan O'Sullivan. Achieving High Performance in Mercurial, July 2006. Europython 2006, 3–5 July 2006. CERN, Geneva.

[50] Kaiping Peng and Richard E. Nisbett. Culture, dialectics, and reasoning about contradiction. *American Psychologist*, 54:741–754, September 1999.

[51] Bruce Perens. *Open Sources: Voices from the Open Source Revolution*, chapter The Open Source Definition. In Dibona et al. [13], February 1999.

[52] Dewayne E. Perry, Harvey P. Siy, and Lawrence G. Votta. Parallel changes in large scale software development: An observational case study. In *International Conference on Software Engineering*, pages 251–260, 1998.

[53] Roger S. Pressman. *Software engineering: a practitioner's approach*. McGraw-Hill, New York, fifth edition, 2001.

[54] Eric S. Raymond. The cathedral and the bazaar. *First Monday*, 3(3), 1998.

[55] Eric S. Raymond. *Open Sources: Voices from the Open Source Revolution*, chapter A Brief History of Hackerdom. In Dibona et al. [13], February 1999.

[56] Ollivier Robert. DVCS or a new way to use Version Control Systems for FreeBSD, May 2006. BSDCan 2006, 12–13 May 2006. Ottawa, Canada.

[57] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, December 1975.

[58] Nathan Rosenberg. Economic experiments. In Giovanni Dosi, David J. Teece, and Josef Chytry, editors, *Understanding industrial and corporate change*, pages 287–308. Oxford university press, 2005.

[59] Walt Scacchi, Joseph Feller, Brian Fitzgerald, Scott A. Hissam, and Karim Lakhani. Understanding free/open source software development processes. *Software Process: Improvement and Practice*, 11(2):95–105, 2006.

[60] Maha Shaikh and Tony Cornford. Version management tools: CVS to BK in the Linux Kernel. In *Taking Stock of the Bazaar: Proceedings of the 3rd Workshop on Open Source Software Engineering*, pages 127–131. ACM, 2003.

[61] Haifeng Shen and Chengzheng Sun. A Complete Textual Merging Algorithm for Software Configuration Management Systems. In *COMPSAC*, pages 293–298, 2004.

[62] Richard Stallman. *Open Sources: Voices from the Open Source Revolution*, chapter The GNU Operating System and the Free Software Movement. In Dibona et al. [13], February 1999.

[63] W. F. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, 1998.

[64] Walter F. Tichy. The string-to-string correction problem with block moves. *ACM Trans. Comput. Syst.*, 2(4):309–321, 1984.

[65] Walter F. Tichy. RCS – A system for version control. *Software-Practice and Experience*, 15(7):637–654, July 1985.

[66] Guido van Rossum. Python reference manual. verkkojulkaisu, September 2006. http://docs.python.org/ref/ref.html (viitattu 3.8.2007).

[67] Ilkka Virtanen. Hiljaisen tiedon ongelma – kuinka hiljaista hiljainen tieto on? Master's thesis, Tampereen yliopisto, Tietojenkäsittelytieteiden laitos, 2006.

[68] Eric von Hippel and Georg von Krogh. Open Source Software and the "Private-Collective"Innovation Model: Issues for Organization Science. *Organization Science*, 14(2):209–223, 2003.

[69] Robert A. Wagner and Michael J. Fischer. The String-to-String Correction Problem. *Journal of ACM*, 21(1):168–173, January 1974.

[70] Juhani Warsta and Pekka Abrahamsson. Is Open Source Software Development Essentially an Agile Method? In *Taking Stock of the Bazaar: Proceedings of the 3rd Workshop on Open Source Software Engineering*, pages 143–147. ACM, 2003.

[71] Bernhard Westfechtel, Bjørn P. Munch, and Reidar Conradi. A layered architecture for uniform version management. *IEEE Trans. Software Eng.*, 27(12):1111–1133, 2001.

[72] Sam Williams. Free as in Freedom – Richard Stallman's Crusade for Free Software, March 2002. http://www.oreilly.com/openbook/freedom/index.html (viitattu 24.8.2007).

[73] Alexander Yip, Benjie Chen, and Robert Morris. Pastwatch: A Distributed Version Control System. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI '06)*, pages 381–394, 2006.

[74] Thomas Østerlie and Letizia Jaccheri. A Critical Review of Software Engineering Research on Open Source Software Development. In *Proceedings of the 2nd AIS SIGSAND European Symposium on Systems Analysis and Design*, Gdansk, Poland, June 2007.

Hakemisto

A GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **"Document"**, below, refers to any such manual or work. Any member of the public is a licensee,

and is addressed as **"you"**. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A **"Modified Version"** of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A **"Secondary Section"** is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The **"Invariant Sections"** are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The **"Cover Texts"** are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A **"Transparent"** copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called **"Opaque"**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a

publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The **"Title Page"** means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section **"Entitled XYZ"** means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as **"Acknowledgements"**, **"Dedications"**, **"Endorsements"**, or **"History"**.) To **"Preserve the Title"** of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work

that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License

does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar

in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright ©YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.