Utilização de Sistemas de Controle de Versão para facilitar o desenvolvimento colaborativo de software

André Silveira Wesley Dal Col

21 de maio de 2010

RESUMO

De modo a atender às diversas demandas do mercado, é cada vez mais comum que o desenvolvimento de software seja realizado 24 horas por dia, por equipes distintas trabalhando de forma descentralizada ao redor do globo. Por isso, são cada vez mais necessárias ferramentas que propiciem o rastreamento e a integração do código fonte. Alguns sistemas de cotrole de versão que foram desenvolvidos para auxiliar na primeira tarefa, evoluíram, tornando-se também boas ferramentas de programação integrado-colaborativa. O foco deste trabalho é extensão do uso dessas ferramentas para auxiliar o desenvolvimento colaborativo entre equipes geograficamente distantes.

Palavras-chave: Controle de Versão, Colaboração, Integração, Desenvolvimento de Software

ABSTRACT

In order to meet various market demands, it is increasingly common for software development is carried out 24 hours a day, by separate teams working in a decentralized way around the globe. Therefore, they are increasingly necessary tools that provide tracking and integration of source code. Some version control systems that were developed to assist in the first task, evolved, becoming too well-integrated programming tools collaboratively. The focus of this paper is to extend the use of these tools to support collaborative development among geographically dispersed teams.

Palavras-chave: Version Control, Colaboration, Integration, Software Development

SUMÁRIO

1 INTRODUÇÃO p. 5

1.1 **Objetivos** . . . . . p. 6

2 HISTÓRICO p. 7

2.1 **A Aurora das Linguagens de Programação Modernas** . . . . . p. 7

2.2 **As ferramentas diff e patch** . . . . . p. 8

2.3 **O Nascimento do Controle de Versões** . . . . . p. 9

2.4 **O Linux e o Controle Descentralizado de Versões** . . . . . p. 10

3 DESENVOLVIMENTO COLABORATIVO p. 12

3.1 **Aplicando patches** . . . . . p. 12

3.2 **Modelos de Ciclo de Trabalho Centralizados** . . . . . p. 13

3.2.1 Lock-Modify-Unlock . . . . . p. 14

3.2.2 Copy-Modify-Merge . . . . . p. 15

3.3 **Modelos de Ciclo de Trabalho Descentralizados** . . . . . p. 15

3.3.1 Integration Manager . . . . . p. 16

3.3.2 Dictator and Lieutenants . . . . . p. 16

3.4 **Colaboração em Paralelo** . . . . . p. 17

REFERÊNCIAS p. 18

1 INTRODUÇÃO

Nos primórdios da computação os computadores eram manipulados apenas por um pequeno grupo de técnicos altamente especializados, que precisavam plugar uma grande quantidade de cabos nos enormes painéis que compunham os computadores da época, num processo lento e muito suscetível a erros.

Com o desenvolvimento da tecnologia e o advento das linguagens e técnicas de programação modernas, a tarefa de criar programas de computador pode ser desempenhada por um número maior de pessoas, muitas vezes, cada uma sendo responsável por apenas uma fração de todo o sistema. Unir todas essas partes e garantir que elas funcionem juntas não é uma tarefa simples. Analogamente a correção manual de erros em grandes sistemas de computação é frequentemente uma tarefa tão dispendiosa quanto criá-lo e mantê-lo funcionando.

Para resolver esses problemas de forma automatizada, com o passar dos anos algumas ferramentas foram sendo desenvolvidas, como os programas diff e patch que facilitam processo de aplicar uma correção em uma determinada parte do software, assim como os Sistemas de Controle de Versão dos quais podemos citar o CVS, o Subversion e Git, que são utilizados para rastrear e controlar a evolução do código fonte, o que torna o trabalho das equipes de desenvolvimento um pouco menos complicado.

Entretanto, a dinâmica atual do mercado de tecnologia gera demandas cada vez mais agressivas, fazendo com que o desenvolvimento de software seja feito quase que 24 horas por dia, muitas vezes por equipes geograficamente distantes ou em fuso-horários muito diferentes, ressurgindo o desafio de integrar vários componentes de software agora produzidos não somente por pessoas distintas mas por empresas e instituições das mais diversas naturezas e culturas, com o máximo de qualidade possível minimizando o tempo de desenvolvimento e os custos.

Com projetos compostos por milhões de linhas de código, torna-se impraticável que a integração entre as partes seja realizada manualmente, pois basta somente um equívoco para provocar erros muito difíceis de corrigir.

Para tentar resolver esse e outros problemas novos Sistemas de Controle de Versão foram

criados, melhorando os mecanismos de mesclagem de código e buscando uma arquitetura descentralizada, de forma que as equipes possam trabalhar com um mínimo de interferência entre si, enquanto a etapa de integração é feita cada vez mais de forma automática.

1.1 Objetivos

O objetivo principal deste trabalho é demonstrar como os Sistemas de Controle de Versão modernos podem ser utilizados para facilitar o desenvolvimento de software colaborativo, explorando duas de suas principais características, a arquitetura descentralizada e a facilidade com que integram peças de código.

Será feito um breve levantamento histórico contextualizando alguns dos principais Sistemas de Controle de Versão em uso atualmente.

Em seguida demostrar-se-á como a arquitetura descentralizada facilita o desenvolvimento de ramificações independentes do código fonte com um mínimo de conflitos entre elas, confrontando esse cenário com situações onde não há ferramenta de controle de versão ou quando a ferramenta utilizada não permite essa abordagem.

Também será mostrado como as novas técnicas de rastreamento da genealogia de um projeto de software permitem uma integração muito mais suave e automática do que quando são utilizadas ferramentas menos sofisticadas ou nenhuma ferramenta.

2 HISTÓRICO

Neste capítulo será abordado brevemente um pouco da história do versionamento de arquivos no contexto do desenvolvimento de software, quais foram as necessidades de cada época e quais as ferramentas desenvolvidas para supri-las, mostrando como foi a evolução deste processo ao longo dos anos.

2.1 A Aurora das Linguagens de Programação Modernas

No século XIX teares programáveis e tocadores automáticos de piano já implementavam o que hoje é conhecido por Linguagem de Domínio Específico (DSN - Domain Specifc Language).

No início do século XX o formalismo dos trabalhos de Alonzo Church – Cálculo Lambda – (CHURCH, 1936) e de Alan Turing – Máquina de Turing – (TURING, 1936) foreneceram as bases matemáticas necessárias para se expressar Algoritmos.

Entretanto, as primeiras linguagens de programação utilizadas por computadores digitais só foram criadas a partir da década de 1940. O termo portabilidade apareceu na década de 1950 e a partir da década de 1960 começaram a ser desenvolvidos os principais paradigmas de programação que conhecemos hoje.

De lá para cá a popularização do uso de computadores fez com que a demanda por software crescesse absurdamente, criando toda uma indústira que movimenta bilões de dólares anualmente.

Acompanhado essas cifras, bilhões de linhas de código tem de ser produzidas, mantidas e atualizadas constantemente, o que nunca foi uma tarefa simples e fica cada vez mais complexa a medida que os sistemas crescem.

2.2 As ferramentas diff e patch

Até a década de 1970, qualquer correção feita em um programa era feita em uma cópia do arquivofonte original (se disponível) e encaminhada ao auto do software para que ele analisasse as alterações e incorporasse no conjunto oficial de fontes.

A tarefa de procurar alterações dentro dos enormes arquivos de código fonte era então realizada manualmente pelo programador, linha a linha, num processo bastante lento.

Em 1974 a primeira versão do programa diff foi liberada juntamente com a quinta versão do sistema operacional Unix, produzido no Bell Laboratories, por Douglas McIlroy, basead em um protótipo escrito por James W. Hunt da Universidade de Stanford. Em 1976 eles publicaram um artigo sobre o algoritmo utilizado (HUNT; MCILROY, 1976).

O programa diff aceita como entrada arquivos ou diretórios e gera como saída apenas as diferenças entre eles especificando os arquivos e números de linhas onde elas ocorrem. O formato da saída ficou conhecido pelo nome diff, e posteriormente patch.

O trabalho de encontrar as diferenças entre os arquivos foi simplificado, mas a tarefa de aplicar as correções ainda era realizada manualmente.

Para resolver esse problema, em maio de 1985 Larry Wall criou o programa patch (WALL, 1985) [1]. Esse programa pega um arquivo no formato gerado pelo diff e aplica no arquivo completo de fontes, automatizando a tarefa de correção. Logo esse processo passou a ser conhecido como patching, onde criar um patch é gerar um arquivo no formato diff e aplicar o petch é usar o programa patch para implantar as alterações.

Esses dois prgramas facilitaram em muito o trabalho de programadores que precisavam gerenciar várias atualizações e correções, entretanto, uma vez aplicado o patch, removê-lo não é tarfa simples. Isso podia acontecer quando por exemplo, um arquivo de diff ou de fonte errados eram utilizados, ou quando a aplicação de um patch corrigia uma falha, mas causava outras.

O aumento do número de patches trouxe outra demanda, a necessidade de saber quem aplicou qual patch e quando, bem como desfazer as alterações caso necessário.

[1] Larry Wall escreveu a versão original do *patch*. Paul Eggert removeu os limites arbitrários do patch; adicionou suporte a arquivos binários, datação e deleção dos arquivos; e deixou-o em conformidade com o padrão POSIX. Outros contribuidores incluem Wayne Davison, que adicionaou suporte ao formato unidiff, e David MacKenzie, que adicionaou configurações e suporte a backup. Andreas Grünbacher adicionou suporte a mesclagem (merging).(WATSON, 2010)

2.3 O Nascimento do Controle de Versões

Uma das mais utilizadas ferramentas de controle de revisão dessa época foi o RCS, criado em 1982 por Walter F. Tichy, sendo uma das primeiras a automatizar tarefas de armazenar, recuperar, identificar e mesclar revisões.

Embora razoavelemente útil para lidar com alguns arquivos de texto o RCS ainda não supria várias necessidades no desenvolvimento de software.

Como um conjunto de scripts trabalhando em conjunto com o RCS, surge em julho de 1986 o CVS, criado por Dick Grune. Em 1989, Brian Berliner o reescreveu completamente em C e Jeff Polk adicionou algumas funcionalidades posteriormente (BAR; FOGEL, 2003).

Mesmo sendo uma ferramenta bastante sofisticada para a época, o CVS ainda mantinha os principais problemas do RCS, como só poder ser utilizado localmente, o que só foi resolvido na década de 1990 por Jim Kingdon.

> I created CVS to be able to cooperate with my students, Erik Baalbergen and Maarten Waage, on the ACK (Amsterdam Compiler Kit) C compiler. The three of us had vastly different schedules (one student was a steady 9-5 worker, the other was irregular, and I could work on the project only in the evenings). Their project ran from July 1984 to August 1985. CVS was initially called cmt, for the obvious reason that it allowed us to commit versions independently.
>
> (GRUNE, 2003)

O CVS foi construído para auxiliar equipes nas quais os programadores tem de trabalhar em horários diferentes, mas um por vez. Quando surge a necessidade de vários programadores trabalharem simultaneamente no mesmo projeto, o design do CVS já não atende tão bem.

Em 1995 Karl Fogel e Jim Blandy criaram a emrpesa Cyclic Software oferecendo suporte ao CVS, e embora a tenham vendido a empresa posteriormente, continuavam a utilizar o CVS no seu dia a dia. Suas frustrações com os problemas do CVS foram tais que Jim começou a pensar em uma melhor forma de lidar com versioanamento.

No início de 2000, a CollabNet contratou Karl para desenvolver um novo sistema de controle de versão para substituir o CVS na sua suite colaborativa CollabNet Enterprise Edition (CEE). Jim então convenceu a empresa onde trabalhava, a Red Hat, a cedê-lo para esse projeto, iniciando assim o desenvolvimento do Subversion (COLLINS-SUSSMAN; FITZPATRICK; PILATO, 2008).

O Subversion foi então um dos primeiros sistemas de controle de versão open source a

pensar desde o seu projeto no desenvolvimento colaborativo.

Embora o Subversion tenha sido desenhado para suprir diversas demandas não satisfeitas pelo CVS, ele ainda era fortemente influenciado pelo seu antecessor, mantendo por exemplo uma arquitetura centralizada, onde a figura de um servidor central era necessária durante a operação.

2.4 O Linux e o Controle Descentralizado de Versões

O Linux é de longe um dos projetos de software livre de maior sucesso na atualidade, e talvez um dos maiores difusores da filosofia open source.

Liberada em 1991 em um post no newsgroup "comp.os.minix." como o resultado de um hobby, a primeira versão do Linux logo cresceu, ganhando novos contribuidores e tornando-se um dos principais sistemas operacionais utilizados em servidores.

> Hello everybody out there using minix -
>
> I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).
>
> I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)
>
> Linus (torvalds@kruuna.helsinki.fi)
>
> PS. Yes – it's free of any minix code, and it has a multi-threaded fs. It is NOT portable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-(.
>
> (TORVALDS, 1991)

Contando com a ajuda de programadores de todas as partes do mundo, logo foi necessário utilizar um sistema de controle de versão capaz de gerenciar toda a complexidade por trás de um desenvolvimento descentralizado.

Para esta tarefa Linus utilizou por vários anos o BitKeeper, ferramenta proprietária mas que ao contrário do CVS e SVN possui uma arquitetura distribuída, facilitando o trabalho de colaboradores espalhados mundo afora.

Mas em 2005 o licenciamento do BitKeeper mudou, tornando o seu uso no projeto Linux inviável. Linus então decidiu criar um sistema de controle de versão descentralizado tomando como base alguns critérios:

1. Tomar o CVS como um exemplo do que não fazer; na dúvida, fazer exatamente a escolha oposta ao que o CVS fez.
2. Suportar um workflow parecido com o do BitKeeper.
3. Fortíssima segurança contra corrupção, acidenta ou maliciosa
4. Altíssima performance

Com essas premissas em mente Linus começou o desenvolvimento do Git e em menos de duas semanas já tinha implementado várias funcionlaidades. Menos de dois meses depois, o kernel 2.6.12 já era lançado tendo seu código gerenciado pelo próprio Git (CHACON, 2009).

3 DESENVOLVIMENTO COLABORATIVO

Na época em que o software se resumia ao conjunto de posições de plugues e cabos, a expressão “trabalho Colaborativo” aplicava-se ao trabalho conjunto de engenheiros mecânicos, eletricistas, técnicos e pesquisadores. Na maioria dos casos a programação primitiva era realizada por pouquíssimas pessoas extremamente especializadas e palavras como “conflitos” não se aplicavam.

Só com o aparecimento das linguagens de alto nível e do uso comercial dos computadores é que começou a haver uma preocupação com a profissionalização da tarefa de programá-los, de forma a possibilitar a produção de software em larga escala.

Mas conforme a quantidade de código ia aumentando, a quantidade pessoas envolvidas no desenvolvimento de uma aplicação, o número de erros de programação e consequentemente de correções e atualizações também crescia.

A análise das falhas e suas correções eram tarefas manuais, demoradas e tediosas, muito propensas a erros, pois uma correção deveria ser propagada até a matriz de fontes de onde eram gerados os binários que eram posteriormente distribuídos.

As primeiras ferramentas para automatizar as tarefas gerenciamento e manutenção de fontes foram o diff e o patch. Alguns anos depois vieram os Sistemas de Controle de Versão centralizados e posteriormente os Sistemas de Controle de Versão distribuídos.

3.1 Aplicando patches

Como visto anteriormente o diff foi criado em 1974 e ajudou a estabelecer diferenças textuais entre arquivos. Em 1985, Larry Wall cia seu complemento, o patch, e desde então a vida dos desenvolvedores foi bastante facilitada.

Para o caso dos softwares cuja distribuição do código fonte era aberta, quando alguém encontrava um problema bastava realizar as correções diretamente no código fonte e recompilá-lo. Para que essas correções chegassem aos desenvolvedores da aplicação, o programador que

fez as correções criava com o programa diff um arquivo contendo as diferenças entre os fontes originais e os fontes corrigidos e enviava aos desenvolvedores originais. Estes aplicavam então as alterações automaticamente utilizando o programa patch. Por esse motivo o arquivo com as diferenças passou a ser conhecido também como “patch” da aplicação.

Esse processo poupava um tempo precioso dos desenvolvedores, que ao invés de gastar semanas analisando toda uma imensidão de fontes, podiam se concentrar apenas nos pontos incongruentes.

Dessa forma o ciclo normal para uma correção de software consistia em:

1. Identificar o problema
2. Identificar nos fontes o trecho de código que gerava o problema
3. Corrigir a falha
4. Criar um patch com as correções utilizando o programa diff
5. Enviar o patch ao mantenedor do software
6. O mantenedor aplicar ou não o patch utilizando o progrma homônimo.

O grande problema agora não era mais localizar e corrigir o problema, mas sim decidir quando aplicar ou não um determinado patch, pois embora seja simples aplicá-lo, reverter um patch problemático é bem mais complicado.

Outro problema difícil de resolver era como manter o rastreamento da genealogia do software. Com tantos patches, responder quem aplicou qual patch e quando, nem sempre era possível.

Esses e outros problemas puderam ser resolvidos com a utilização dos Sistemas de Controle de Versão, que além de facilitar ainda mais a aplicação de patches, auxiliam nas tarefas de rastreamento, bifurcação e mesclagem de código.

Conforme essas ferramentas foram evoluindo, outros ciclos de trabalho foram sendo adotados, de forma a simplificar o processo de desenvolvimento de software como veremos adiante.

3.2 Modelos de Ciclo de Trabalho Centralizados

Com o advento dos Sistemas de Controle de Versão, as equipes puderam estabelecer ciclos de trabalho mais robustos, utilizando-se das facilidades fornecidas por esses softwares.

Esses ciclos de trabalho são nada mais que convenções a respeito da sequência de passos necessários para manter o bom andamento das tarefas de desenvolvimento.

Embora isso tenha ajudado padronizar as formas de se trabalhar, essas metodologias ainda eram fortemente limitadas pelo que os SCV conseguiam fazer.

Alguns do ciclos de trabalho tiveram maior destaque no mercado e foram perpetuando-se aolongo dos anos. Os mais conhecidos e uilizados serão explicados brevemente.

3.2.1 Lock-Modify-Unlock

No modelo de desenvolvimento Lock-Modify-Unlock (Travar-Modificar-Liberar) o desenvolvedor toma posse do arquivo ou conjunto de arquivos travando-os, de forma nenhum outro membro do projeto possa modificá-los. Então ele realiza as edições que achar necessárias e aplica testes. Quando concluir as alterações ele então libera os arquivos para que outras pessoas possam aterá-los.

A utilização dessa metodologia foi muito comum com o RCS e o CVS, e também está disponível no Subversion. Ela impede que mais de uma pessoa edite o mesmo arquivo simultaneamente. O ciclo padrão de desenvolvimento consiste basicamente em:

1. Travar o arquivo no servidor
2. Realizar as alterações e testes que forem necessários
3. Destravar o arquivo no servidor.

Esse tipo de ciclo de trabalho mostra-se adequado em projetos onde os arquivos são razoavelmente independentes uns dos outros, podendo ser alterados separadamente sem maiores prejuízos aos restantes, pois, quanto maior a quantidade de arquivos travados, maior as chances de outro desenvolvedor precisar esperar enquanto as atualizações não ficam prontas.

Se por um lado isso evita conflitos, por outro adiciona problemas pois se um desenvolvedor se esquecer de liberar um arquivo anteriromente travado por ele, ou pior ainda, se esse desenvolvedor sair de férias, esses arquivos permanecerão indisponíveis até que ele retorne ou até que um administrador libere os arquivos manualmente (COLLINS-SUSSMAN; FITZPATRICK; PILATO, 2008).

3.2.2 Copy-Modify-Merge

O modelo Copy-Modify-Merge (Copiar-Modificar-Mesclar) é o modelo de ciclo de desenvolvimento padrão quando se trabalha com o Subversion. Com ele cada desenvolvedor extrai do servidor uma cópia dos arquivos com os quais pretende trabalhar. Em seguida ele realiza as alterações que julgar necessárias e as publica no servidor, mesclando seu trabalho sobre sua cópia ao conteúdo do arquivo remoto.

Quando algum desenvolvedor fizer uma cópia dos dados no servidor, as alterações criadas por outros desenvolvedores são automaticamente propagadas.

Problemas começam a surgir quando mais de um desenvolvedor precisa alterar o mesmo arquivo simultaneamente, principalmente em trechos muito próximos entre si. Na maioria das vezes o próprio Sistema de Controle de Versão cuida para que se possam fazer mesclagens automáticas. Quando isso não é possível, temos o que chamanos de conflitos. Estes devem ser resolvidos manualmente pelos desenvolvedores.

O ciclo normal de desenvolvimento é portanto:

1. Extrair uma cópia dos arquivos no servidor
2. Realizar as alterações e testes necessários
3. Verificar e resolver conflitos, se existirem
4. Publicar as alterações no servidor

3.3 Modelos de Ciclo de Trabalho Descentralizados

Embora a evolução dos Sistemas de Controle de Versão tenham propiciado até este ponto a solução de vários problemas, basta que o servidor fique indisponível, por exemplo, para que todo o ciclo de desenvolvimento fique comprometido.

Projetos maiores e com equipes localizadas geograﬁamente distantes necessitam de algumas outras funcionalidades. A principal delas é que o ciclo de desenvolvimento seja descentralizado.

Alguns dos Sistemas de Controle de Versão que permitem nativamente esse tipo de ciclo de trabalho são o Git, o Mercurial e o Bazaar.

3.3.1 Integration Manager

Neste cenário, cada desenvolvedor possui uma cópia do repositório público, onde ele realiza suas alterações. Quando se fizer necessário, o desenvolvedor avisa a um Gerente de Integração que há código disponível em sua área pública. O gerente então avalia e absorve absorve essas alterações e publica-as no repositório principal da empresa. Cada desenvolvedor é então responsável por sincronizar seu repositório particular com o repositório partcular da empresa e repete-se o ciclo(CHACON, 2009).

O fluxo de trabalho é parecido com:

1. O desenvolverdor sincroniza seu repositório particular com o da empresa
2. Realiza as alterações e testes necessários
3. Comunica ao Gerente de Integração que há código disponível
4. O Gerente avalia e atualiza ou não o repositório central da empresa

3.3.2 Dictator and Lieutenants

Este ciclo de desenvolvimento é bem parecido com o anterior e é o modelo adotado no desenvolvimento do kernel do Linux atualmente.

Nele o "ditador" Linus Torvalds é responsável por publicar a árvore oficial de fontes do kernel, que é autoritativa em relação às demais. Ele aceita atualizações apenas de alguns "tenentes confiáveis" (O'SULLIVAN, 2009), que por sua vez são responsáveis por algum dos subsistemas do kernel.

Esses "tenentes" podem possuir um ciclo de trabalho completamente diferente do "ditador", ou ainda podem ser ditadores de outros tenentes, adicionando camadas de hierarquia completamente heterogêneas. Dessa forma, cada equipe pode trabalhar internamente seguindo alguns procedimentos, enquanto mantém uma interface bem definida com os outros grupos.

Aproveitando-se de técnicas mais eficientes para ramificação e mesclagem de código, o ciclo usual de trabalho nesse modelo de desenvolvimento é:

1. Atualizar seu repositório com as atualizações publicadas pelo ditador
2. Realizar as alteraçẽos e testes necessários

3. Submeter essas alterações à avaliação do seu superior imediato na hieraquia, que pode incluí-las no no código oficial um nível acima, ou rejeitá-las.

3.4 Colaboração em Paralelo

Os Sistemas de Controle de Versão e modelos de desenvolvimento centralizados mostram-se mais adequados a pequenos projetos ou a situações onde mesmo uma equipe grande localiza-se geograficamente próxima, pois depende de uma boa comunicação e da disponibilidade constante do servidor central.

Como vimos, basta que o servidor fique indisponível nesses casos, que os Sistemas de Controle de Versão centralizados mostram-se indeficazes, muitas vezes até atrapalhando o ciclo normal de desenvolvimento.

Já os Sistemas de Controle de Versão e ciclos de trabalho descentralizados, permitem que células independentes realizem seu trablaho em paralelo, de forma bastante flexível, mesmo sob condições adversas de comunicação.

No desenvolvimento deste trablaho mostraremos alguns estudos de caso que demonstram como essas ferramentas originalmente desenvolvidas somente para controle e rastreamento de versões podem ser aplicadas para otimizar a performance de equipes em situações geográficas distintas.

REFERÊNCIAS

BAR, M.; FOGEL, K. *Open Source Development with CVS*. 3. ed. Scottsdale, AZ: Paraglyph Press, 2003.

CHACON, S. *Pro Git*. 1. ed. New York, NY: Apress, 2009.

CHURCH, A. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, v. 58, n. 2, p. 345–363, Abril 1936.

COLLINS-SUSSMAN, B.; FITZPATRICK, B. W.; PILATO, C. M. *Version Control with Subversion*. 2. ed. Cambridge, MA: O'Reilly Media, 2008.

GRUNE, D. *Dick Grune's website*. 2003. Disponível em: <`http://www.cs.vu.nl/~dick/CVS.html#History`>.

HUNT, J. W.; MCILROY, M. D. An algorithm for differential file comparison. *Computing Science Technical Report*, Bell Laboratories, Junho 1976.

O'SULLIVAN, B. *Mercurial: The Definitive Guide*. 1. ed. Cambridge, MA: O'Reilly Media, 2009.

TORVALDS, L. *What would you like to see most in minix?* Usenet group comp.os.minix: [s.n.], Agosto 1991. Disponível em: <`http://groups.google.com/group/comp.os.minix/msg/b813d52cbc5a044b`>.

TURING, A. M. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, London, 1936.

WALL, L. *patch version 1.3*. Usenet group mod.sources: [s.n.], Maio 1985. Disponível em: <`http://groups.google.com/group/mod.sources/msg/488b0929254d936a`>.

WATSON, C. *Unix Online Manual*. [S.l.], Fevereiro 2010.