UNIVERSIDADE FEDERAL DE JUIZ DE FORA

Departamento de Ciência da Computação

Análise Comparativa entre Sistemas de Controle de Versões

Daniel Tannure Menandro de Freitas

JUIZ DE FORA

DEZEMBRO, 2010

Análise Comparativa entre Sistemas de Controle de Versões

Daniel Tannure Menandro de Freitas

Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Orientador: Profa. Alessandreia Marta de Oliveira

JUIZ DE FORA
DEZEMBRO, 2010

ANÁLISE COMPARATIVA ENTRE SISTEMAS DE CONTROLE DE VERSÕES

Daniel Tannure Menandro de Freitas

MONOGRAFIA SUBMETIDADA AO CORPO DOCENTE DO DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

_____

Profa. Alessandreia Marta de Oliveira - orientadora
M. Sc. em Engenharia de Sistemas, COPPE/UFRJ, 2003

_____

Prof. Rubens de Oliveira
Doutor em Engenharia Civil, PUC-Rio, 1994

_____

Prof. Jairo Francisco de Souza
M. Sc. em Engenharia de Sistemas, COPPE/UFRJ, 2003

JUIZ DE FORA, MG – BRASIL

DEZEMBRO, 2010

AGRADECIMENTOS

Agradeço à minha família, por me apoiar em todos os momentos.

À minha orientadora, prof. Alessandreia, pela excelente orientação.

Ao prof. Rubens, por me permitir fazer parte, desde o começo, de todo o avanço que o Instituto de Ciências Exatas da UFJF obteve com a sua direção.

Aos colegas de curso e do Grupo de Educação Tutorial da UFJF, que realiza pesquisas na área de Gerência de Configuração e é coordenado pela prof. Alessandreia, Rafael Barros, Michelle Brum e Leonardo Costa, que me ajudaram no processo de revisar e corrigir erros nesta monografia.

A Deus por me dar saúde para alcançar meus objetivos.

Resumo

Os sistemas de controle de versões são responsáveis pela identificação de itens de configuração e devem permitir que eles evoluam de forma distribuída, concorrente e disciplinada. Isso é necessário para que as solicitações de alteração ocorram paralelamente e não comprometam o andamento do projeto. Existem muitos sistemas de controle de versões disponíveis e saber quais são as vantagens e desvantagens de cada um se torna, cada vez mais, uma tarefa complexa. Este trabalho apresenta um comparativo entre três ferramentas de controle de versões; Subversion, Git e Mercurial, abordando as principais características de cada uma dessas ferramentas.

Palavras-chave: Sistemas de Controle de Versão, Gerência de Configuração de Software, Subversion, Git, Mercurial

Abstract

Version control systems are responsible for identifying item configurations and allowing them to evolve in distributed, concurrent and disciplined manners. This is necessary to allow change requests to occur simultaneously and to not compromise the project's progress. There are many version control systems available and determining what are the advantages and disadvantages of each one becomes increasingly complex. This paper presents a comparison between three version control tools; Subversion, Git and Mercurial, discussing the main features of each system.

Keywords: *Version Control System, Software Configuration Management, Subversion, Git, Mercurial*

Lista de Siglas

ASF	*Apache Software Foundation*
CVS	*Concurrent Versions System*
DFSG	*Debian Free Software Guidelines*
GC	Gerência de Configuração
GCS	Gerência de Configuração de Software
GPL	*General Public License*
GUI	*Graphical User Interface*
IC	Item de Configuração
RCS	*Revision Control System*
SCCS	*Source Code Control System*
SCV	Sistema de Controle de Versões
VID	*Version Identifier*
IDE	*Integrated Development Environment*

Lista de Figuras

Lista de Tabelas

SUMÁRIO

1. INTRODUÇÃO

A competitividade no mercado de software exige que as empresas vivam um processo de transformação contínua e muitas vezes gerenciar essas transformações não é trivial, dada a dinâmica dos componentes envolvidos em um projeto de software.

São vários os problemas que afetam de forma crônica a indústria de software, como o não cumprimento de prazos e orçamentos equivocados, tornando-se comum o emprego de métodos improvisados para tentar saná-los. Soma-se a esses problemas o fato de muitos desenvolvedores considerarem o software como uma forma de arte e não de engenharia, o que faz com que os processos de desenvolvimento de software sejam dependentes de talentos e esforços individuais, aumentando consideravelmente a chance de erros no projeto.

Ao longo do ciclo de vida de um projeto de software uma grande quantidade de itens é produzida. A probabilidade desses itens sofrerem alterações, devido a mudanças nos requisitos ou correções de defeitos, por exemplo, é muito alta. É comum também que sejam geradas diferentes versões do software à medida que novos problemas sejam descobertos e resolvidos. Controlar e gerenciar todas essas mudanças é essencial para que o projeto seja bem sucedido.

Segundo Pressman (1995), Engenharia de Software é "o estabelecimento e uso de sólidos princípios de engenharia para que se possa obter economicamente um software que seja confiável e que funcione eficientemente em máquinas reais".

A Gerência de Configuração de Software (GCS) é um importante campo da engenharia de software e vem sendo adotada de forma mais ampla a cada dia para apoiar estas questões.

A GCS pode ser definida como a disciplina que permite manter sob controle a evolução dos sistemas de software, gerenciando e rastreando mudanças e restrições de qualidade e cronograma (ESTUBLIER, 2000).

Segundo Murta (2006), sob a perspectiva de desenvolvimento, a GCS é dividida em três sistemas principais, que são: controle de modificações, controle de versões e gerenciamento de construção.

O Sistema de Controle de Versões (SCV) consiste, basicamente, em um local para armazenamento de artefatos gerados durante o desenvolvimento de sistemas de software (MASON, 2006).

Existem muitos sistemas de controle de versões disponíveis, cada um com características e funcionalidades particulares. O grande número de opções disponíveis, aumenta a probabilidade de que se consiga encontrar uma ferramenta que atenda às necessidades do projeto, porém, com tantas ferramentas distintas entre si, o trabalho de comparação entre elas, visando encontrar a melhor para cada caso, torna-se complexo.

O objetivo deste trabalho é auxiliar no processo de escolha de uma ferramenta de controle de versões, estabelecendo parâmetros de comparação e analisando algumas das ferramentas mais populares.

Esses parâmetros foram selecionados de acordo com uma perspectiva gerencial, visando não só comparar características de operações de repositório, mas também, a documentação disponível, a portabilidade, clientes gráficos disponíveis, e outras características relevantes.

Foram selecionados três sistemas de controle de versões: Subversion, Git e Mercurial. Para compará-los, foram abordadas as principais características de um SCV. Essas características serviram de base para apresentar uma visão geral de cada ferramenta e também, para a elaboração da tabela 6.1, que apresenta o comparativo final entre as três ferramentas.

As características selecionadas com o intuito de comparar e descrever uma visão geral das ferramentas são:

- a licença adotada,
- a compatibilidade com os principais sistemas operacionais,
- a interoperabilidade com IDEs (*Integrated Development Environment*),
- a documentação disponível,
- a facilidade de implantação,
- a integração de rede,
- a atomicidade da operação de *commit*,
- a replicação remota de repositório,
- a propagação de alterações para outros repositórios,
- o gerenciamento de permissões,
- as informações, linha por linha, de alterações em um arquivo,

- o suporte à *changesets*,
- o suporte a mesclagem inteligente após renomeações,
- o suporte a movimentações e renomeações de arquivos e diretórios,
- o suporte a cópias de arquivos e diretórios,
- o monitoramento de alterações locais,
- o suporte à atribuição de mensagens de *log* a cada arquivo em um *commit*,
- o suporte à *check out* parcial,
- a disponibilidade de ferramentas de interface web e
- a disponibilidade de clientes de interface gráfica para a ferramenta.

Para cumprir tal objetivo, esta monografia está organizada em mais cinco capítulos, além deste primeiro de introdução. Os capítulos 3, 4 e 5 foram todos organizados com base nas características listadas aqui, e por isso, possibilitam uma leitura transversal.

O capítulo 2 apresenta uma introdução à GCS, apresentando um pequeno histórico de desenvolvimento da área, descrevendo suas perspectivas, funções e sistemas de apoio a cada função. Este capítulo também apresenta uma introdução aos principais conceitos de SCV, apresentando um histórico de desenvolvimento dos SCVs, além de alguns conceitos e modelos.

Os capítulos 3, 4 e 5 apresentam uma visão geral sobre os SCVs Subversion, Git e Mercurial, respectivamente. Estes capítulos descrevem um histórico de desenvolvimento de cada ferramenta e uma visão geral sobre elas. As seções 3.3, 4.3 e 5.3, descrevem uma visão geral de cada ferramenta e foram organizadas de acordo com as características de comparação presentes neste capítulo 1.

Finalmente, o capítulo 6 conclui essa monografia, apresentando as suas principais contribuições e enumerando possíveis trabalhos futuros.

2 GERÊNCIA DE CONFIGURAÇÃO DE SOFTWARE

Este capítulo apresenta uma visão geral sobre Gerência de Configuração (GC), na seção 2.1. A seção 2.2 apresenta uma visão geral sobre Gerência de Configuração de Software, apresentando seu histórico e duas perspectivas de análise: gerencial e de desenvolvimento. A seção 2.3 apresenta os principais conceitos e características de um Sistema de Controle de Versões. Um estudo mais aprofundado sobre SCV pode ser encontrado em (CONRADI e WESTFECHTEL, 1998) e (ESTUBLIER et al., 2002).

2.1 GERÊNCIA DE CONFIGURAÇÃO

Segundo Hass (2003) a GC pode ser analisada sob seis diferentes perspectivas dentro de uma empresa. Sob a perspectiva da pessoa, a GC é categorizada em regras de gerência de configuração, organizacionais, relativas a projetos e externas, que são criadas e seguidas pelas pessoas. Sob a perspectiva do produto a GC depende da natureza do mesmo, levando em conta características como se o produto é simples ou complexo, se tem impacto direto na vida das pessoas ou se é de alto risco por exemplo. A perspectiva de projeto esta relacionada com o desempenho do gerenciamento de configuração para o desenvolvimento de um produto, em um ou vários projetos, durante todo o seu ciclo de vida. A perspectiva inter-organizacional destaca o fato de todas as empresas terem objetos inter organizacionais, como a infra-estrutura, bens da empresa e a documentação da empresa, onde a GC também possui papel importante.

Ainda segundo Hass (2003), a perspectiva de processo considera o fato de que logo que uma empresa começa a considerar o gerenciamento de configuração, a perspectiva de processo precisa ser levada em conta, pois para sustentar o trabalho, os processos devem ser entendidos e implementados e devem sofrer uma melhoria contínua, por isso, praticamente todos os modelos de maturidade destacam a importância da GC. A perspectiva de ferramentas considera que é praticamente impossível lidar com gerência de configuração sem ferramentas adequadas para tal, e por isso, muitas ferramentas estão disponíveis no mercado.

A GC se refere à gestão de um arranjo relativo das partes ou elementos e tem sido cada vez mais utilizada nos processos de desenvolvimento de software.

2.2 GERÊNCIA DE CONFIGURAÇÃO DE SOFTWARE

A Gerência de Configuração surgiu nos anos 50, devido principalmente à indústria da guerra (LEON, 2000). O surgimento da Gerência de Configuração de Software se deu nos anos 60 e 70, quando além dos artefatos de hardware, artefatos de software também passaram a ser abrangidos pela GC (CHRISTENSEN e THAYER, 2002). Foi somente nos anos 80 que a GCS deixou de se restringir às aplicações militares e passou a ser aplicada em organizações não militares (ESTUBLIER et al., 2005).

Segundo IEEE (1990), a GCS é "uma disciplina que aplica procedimentos técnicos e administrativos para identificar e documentar as características físicas e funcionais de um Item de Configuração (IC), controlar as alterações nessas características, armazenar e relatar o processamento das modificações e o estágio da implementação e verificar a compatibilidade com os requisitos especificados".

A Gestão de Configuração de Software é então a arte de coordenar o desenvolvimento de software, identificando, organizando e controlando modificações de software. A GCS também pode ser definida como o desenvolvimento e aplicação de padrões e procedimentos para gerenciar um produto de sistema em desenvolvimento (SOMMERVILLE, 2003)

A GCS pode ser tratada sob a perspectiva gerencial e sob a perspectiva de desenvolvimento, conforme a Figura 2.1:

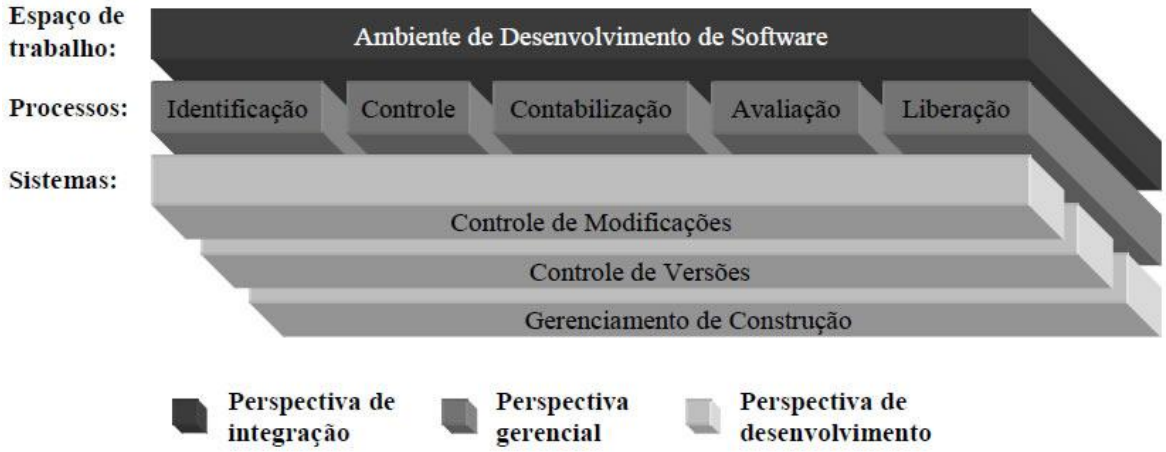


FIGURA 2.1 – Perspectivas de GCS (MURTA, 2006)

A perspectiva gerencial é dividida em cinco pontos, que são (IEEE, 2005):

- identificação da configuração, que define, nomeia, numera e descreve os ICs;
- função de controle da configuração, que acompanha a evolução dos ICs selecionados e descritos pela função de identificação;
- função de contabilização da situação da configuração, que armazena e permite o acesso a todas as informações geradas pelas outras funções;
- função de avaliação e revisão da configuração, que compreende a auditoria funcional e física da linha base;
- função de gerenciamento de liberação e entrega que é responsável pela construção e liberação do ICs além da entrega dos produtos de software.

A perspectiva de desenvolvimento é dividida em três diferentes sistemas, que são (MURTA, 2006):

- **sistema de controle de versões**, que deve permitir a identificação dos ICs e a evolução dos mesmos, de forma distribuída, concorrente e disciplinada. Ele é o responsável por cuidar que as diversas solicitações de modificação possam ser tratadas em paralelo, sem corromper o sistema de GCS como um todo,
- **sistema de controle de modificações**, que é responsável pelo controle da configuração, armazenando as informações das solicitações de modificação e relatando essas informações aos outros participantes e o
- **sistema de gerenciamento de construção**, que é responsável por automatizar o processo de transformação dos artefatos de software em um arquivo executável e estruturar as linhas bases selecionadas para a liberação.

2.3 SISTEMA DE CONTROLE DE VERSÕES

2.3.1 HISTÓRICO

Segundo Bolinger e Bronson (1995), o primeiro sistema de controle de versões foi desenvolvido em 1972, no laboratório Bell Labs, por Marc. J. Rochkind e se chamava Source Code Control System (SCCS). Apesar de ser considerado obsoleto, ele foi o principal sistema de controle de versão até o surgimento do RCS (Revision Control System) e uma de suas principais contribuições foi uma técnica de armazenamento chamada *interleaved deltas*, considerada por vários desenvolvedores de SCVs como a chave para o surgimento de técnicas de junção.

O RCS foi desenvolvido em 1982 por Walter F. Tichy, utilizando técnicas mais avançadas do que o SCCS e ainda é mantido pelo GNU Project. Segundo Spain (2001), o CVS *(Concurrent Versions System*) foi desenvolvido com base no RCS, mas com a possibilidade de gerenciar projetos inteiros e não só um arquivo individualmente, como no RCS. Seu projeto foi iniciado em 1986 por Dick Grune e tinha o nome de CMT. O atual CVS, que se tornou muito popular, teve início com Brian Berliner em 1989. O Subversion, desenvolvido, pela empresa CollabNet, foi desenvolvido com a proposta de melhorar as funcionalidades do CVS e também obteve uma grande aceitação no mercado (SUSSMAN, FITZPATRICK e PILATO, 2009).

Paralelamente ao desenvolvimento e surgimento de novas ferramentas de controle de versão, que trabalhavam de forma centralizada, a SUN começou a desenvolver o TeamWare, para controlar projetos internos da empresa. Ele trabalhava de forma distribuída e posteriormente passou a não ser mais utilizado, devido à comunidade de desenvolvedores da SUN ter optado por softwares que trabalhavam com o mesmo conceito de distribuição, mas com funcionalidades mais modernas, como o Mercurial. Dos sistemas de controle de versão que trabalham de forma distribuída, dois dos que mais ganharam adeptos e tiveram maior aceitação, foram o Mercurial, criado por Matt Mackall e o Git, criado por Linus Torvalds, o também criador do sistema operacional Linux.

A Figura 2.2 mostra os principais sistemas de controle de versão, divididos em livres, que categoriza softwares gratuitos e/ou de código aberto, e comerciais, que categoriza softwares proprietários e pagos.

Centralizado		Distribuído	
Livre	Comercial	Livre	Comercial
SCCS(1972)	CCC/Harvest(1977)	GNU arch(2001)	TeamWare(199?)
RCS(1982)	ClearCase(1992)	Darcs(2002)	Code co-op(1997)
CVS(1990)	Sourcesafe(1994)	DCVS(2002)	BitKeeper(1998)
CVSNT(1998)	Perforce(1995)	SVK(2003)	Plastic SCM(2006)
Subversion(2000)	TFS(2005)	Monotone(2003)	
		Codeville(2005)	
		Git(2005)	
		Mercurial(2005)	
		Bazaar(2005)	
		Fossil(2007)	

FIGURA 2.2 – Sistemas de Controle de Versões

2.3.2 CONCEITOS

Apesar de terem sido desenvolvidas várias ferramentas de controle de versões, existem alguns termos que são comuns a todas essas ferramentas, pois são características de qualquer SCV. Os principais termos e características encontrados em SCVs são:

- **item de configuração**: que representa cada um dos elementos de informação que são criados, ou que são necessários, durante o desenvolvimento de um produto de software. Eles devem ser identificados de maneira única e sua evolução deve ser passível de rastreamento,
- **repositório**: que é o local de armazenamento de todas as versões dos arquivos,
- **versão**: que representa o estado de um item de configuração que está sendo modificado. Toda versão deve possuir um identificador único, ou VID (V*ersion Identifier*),
- **revisão**: que é uma versão que resulta de correção de defeitos ou implementação de uma nova funcionalidade. As revisões evoluem

seqüencialmente. Na figura 2.3 é possível observar que a versão 2 substitui a versão 1, por isso, a versão 2 é um revisão da versão 1,

- **ramo**: que é uma versão paralela ou alternativa. Os ramos não substituem as versões anteriores e são usados concorrentemente em configurações alternativas. Na figura 2.3 a versão 3.1 é uma versão alternativa à versão 3, por isso, ela é classificada como um ramo da versão 3,

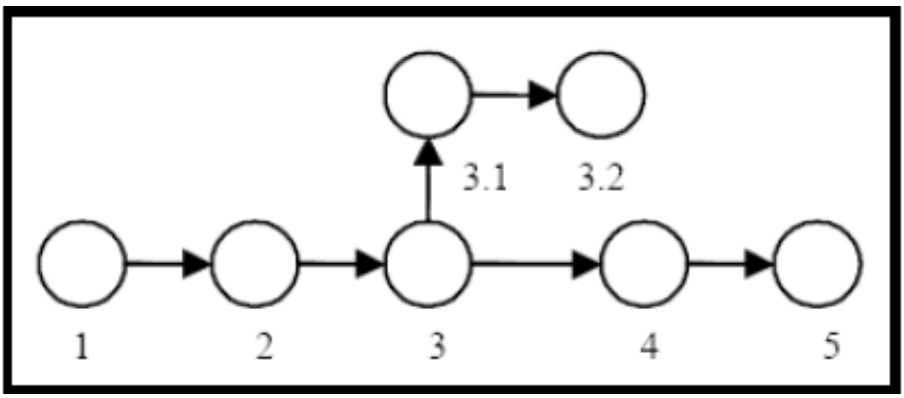


FIGURA 2.3 – Ramificações de Versões (ESTUBLIER et al., 2002)

- **espaço de trabalho**: que é o espaço temporário para manter uma cópia local da versão a ser modificada. Ele isola as alterações feitas por um desenvolvedor de outras alterações paralelas, tornando essa versão privada,
- ***check out*** **(clone)**: que é o ato de criar uma cópia de trabalho local do repositório,
- ***update***: que é o ato de enviar as modificações contidas no repositório para a área de trabalho,
- ***commit*****:** que é o ato de criar o artefato no repositório pela primeira vez ou criar uma nova versão do artefato quando este passar por uma modificação,
- ***merge***: que é a mesclagem entre versões diferentes, objetivando gerar uma única versão que agregue todas as alterações realizadas. Para a realização da mesclagem são utilizados algoritmos que diferenciam os itens de configuração em questão, o que caracteriza o conceito de *delta*. Deltas são responsáveis por economizar espaço e memória, já que, em geral, 98% das duas versões são iguais e apenas os 2% restantes é que serão armazenados (ESTIBLIER, 2000) e

- ***changeset***: que é uma coleção atômica de alterações realizadas nos arquivos do repositório.

No SCV o repositório vai armazenar todo o histórico de evolução do projeto e o desenvolvedor deve copiar os arquivos do repositório para sua área de trabalho, onde ele pode efetuar as modificações desejadas e salvar os arquivos novamente no repositório.

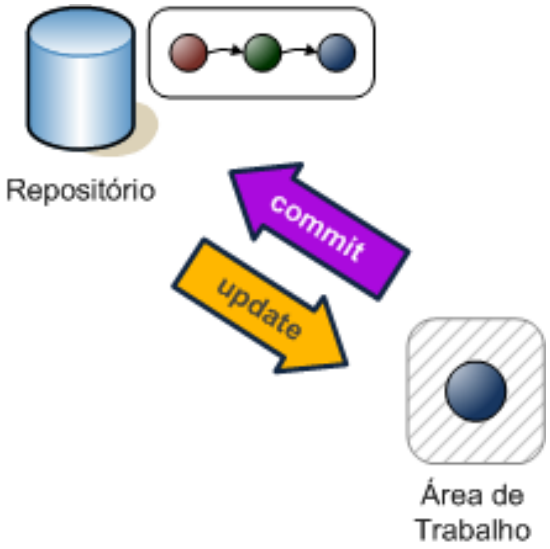


FIGURA 2.4 – Comunicação entre repositório e área de trabalho (DIAS, 2009)

2.3.3 MODELO CENTRALIZADO E DISTRIBUÍDO

Os sistemas de controle de versões podem ser classificados em dois modelos de gerenciamento de repositórios: centralizado e distribuído.

No modelo centralizado existe apenas um repositório central e várias cópias de trabalho. Neste modelo as operações de *commit* e *update* acontecem entre cliente e servidor.

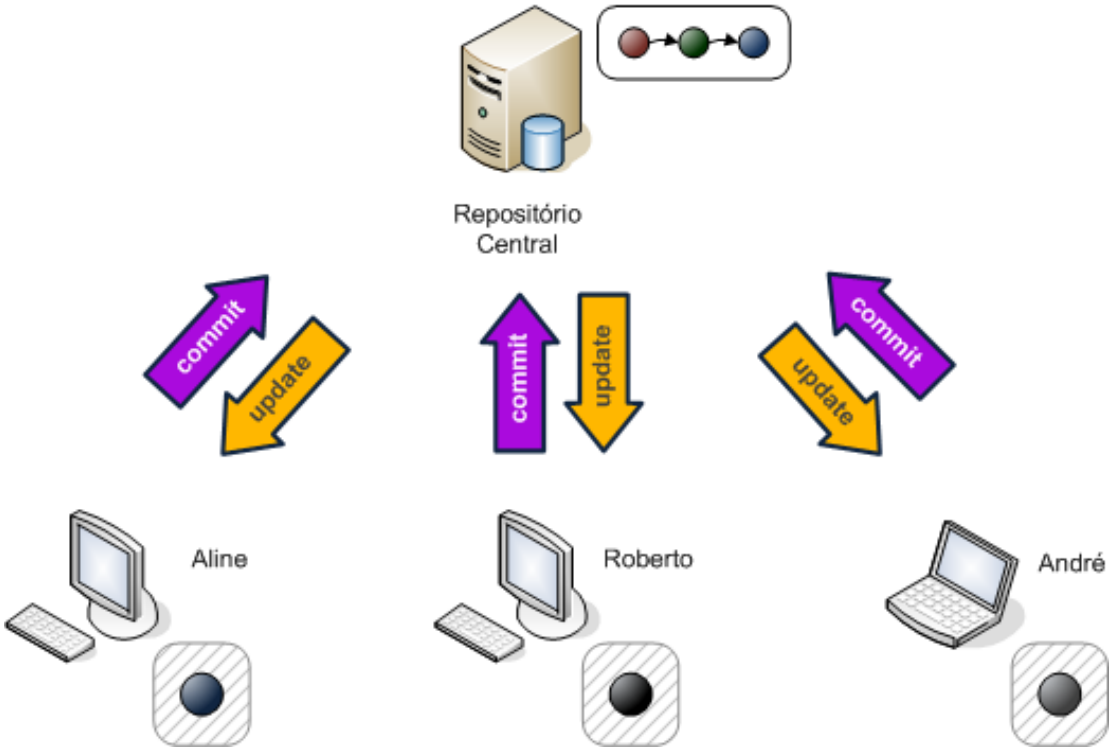


FIGURA 2.5 – SCV – Modelo Centralizado (DIAS, 2009)

A rotina básica de um desenvolvedor que trabalha com um sistema de controle de versões centralizado começa pela execução do comando *check out*, onde é carregada uma cópia de trabalho no local especificado. Para trazer as alterações do repositório para sua área de trabalho, o desenvolvedor deve executar o comando *update*. Após realizar as modificações desejadas, o desenvolvedor executa o comando *commit*, que é responsável por enviar essas modificações ao repositório. Entre a execução dos comandos *commit* e *update*, podem ocorrer conflitos entre os arquivos manipulados por essas operações e para sanar tal problema deve-se utilizar o comando *merge*, que faz uma mesclagem entre os arquivos, resolvendo o problema.

No modelo distribuído, ou descentralizado, existem vários repositórios autônomos e independentes, um para cada desenvolvedor, e cada um desses repositórios possui uma área de trabalho acoplada a ele. Neste modelo as operações de *commit* e *update* acontecem localmente.

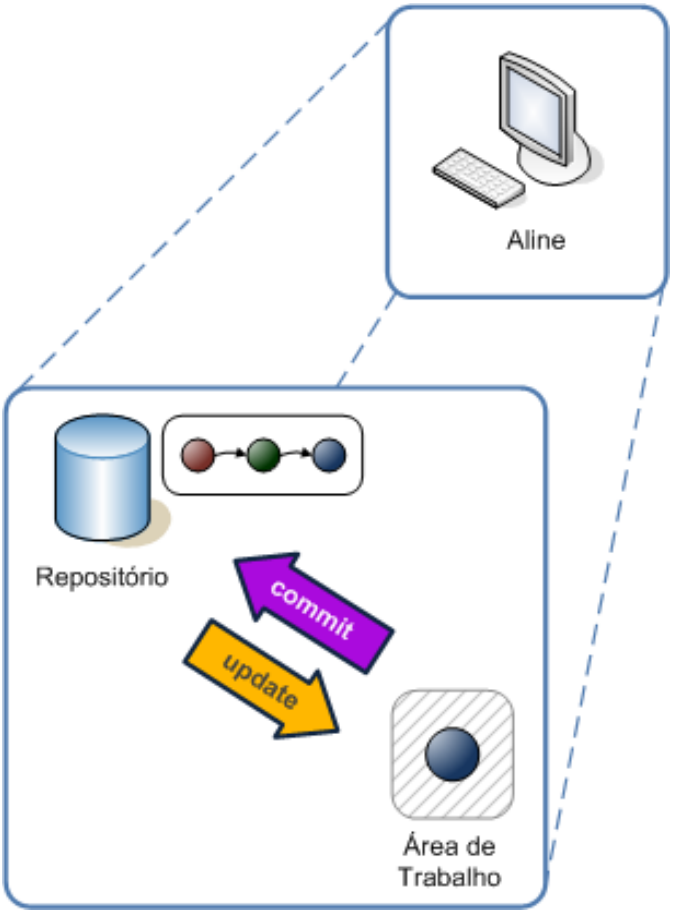


FIGURA 2.6 – SCV – Modelo Distribuído (DIAS, 2009)

A rotina básica de um desenvolvedor que trabalha com um sistema de controle de versões distribuído é semelhante à rotina do desenvolvedor que trabalha com o sistema de modelo centralizado. As diferenças básicas são que, no modelo distribuído o comando *check out* é chamado de *clone*, as operações de *commit* e *update* acontecem localmente e é possível realizar a sincronização de repositórios através dos comandos p*ull* e p*ush*. O comando *pull* atualiza o repositório local (destino) com as alterações realizadas em outro repositório (origem) e o comando *push* envia as alterações do repositório local (origem) para outro repositório (destino).

Muitos projetos de software que utilizam ferramentas de controle de versões descentralizadas adotam também a topologia cliente-servidor, utilizando um servidor central. Isso facilita e simplifica o fluxo de trabalho dos usuários. Existem muitos sites que oferecem hospedagem para projetos que utilizam SCV descentralizados, fazendo o papel

do servidor central. A figura 2.7 mostra como podem ser feitas as propagações entre os usuários e o servidor, adotando esta topologia com um SCV descentralizado.

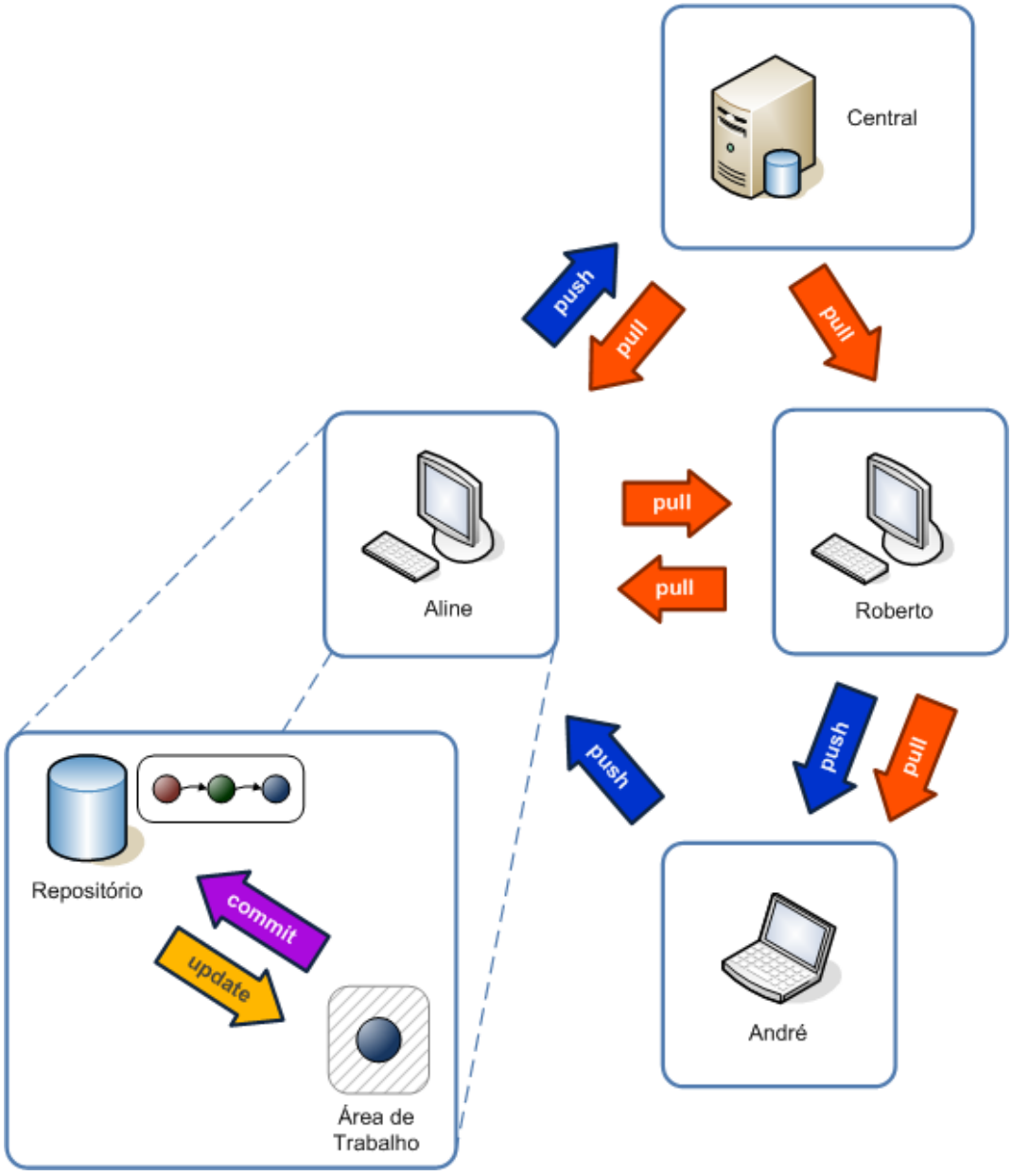


FIGURA 2.7 – Utilização de SCV descentralizado com um repositório central (DIAS, 2009)

2.4 CONCLUSÃO

A Gerência de Configuração de Software é essencial para manter a estabilidade na evolução do projeto e são muitas as ferramentas de apoio à GCS. A maior parte dessas ferramentas é gratuita e/ou de código aberto, por isso, independente do porte de uma

empresa, o uso de algumas dessas ferramentas é totalmente viável e altamente recomendado.

Além do desenvolvimento de muitas ferramentas, a GCS ganhou notoriedade pela sua inclusão nos principais modelos de maturidade, como o CMMI.

Apesar dos sistemas de controle de versões que trabalham com o modelo distribuído terem surgido mais recentemente e terem conseguido uma aceitação tão rápida no mercado, em muitos casos ainda é melhor e mais viável o uso de sistemas que trabalham com o modelo centralizado.

O uso de SCVs tem se tornado comum em muitas empresas, devido ao baixo custo de implantação, o grande número de ferramentas e a vasta documentação disponível.

Para que o sistema de GCS seja implantado como um todo, é importante que ferramentas de controle de mudanças e gerenciamento de construção também sejam estudadas e selecionadas de acordo com as necessidades de cada projeto.

3 SUBVERSION

Neste capítulo é apresentada a primeira das três ferramentas de controle de versões que comparadas, o Subversion. O Subversion é um SCV centralizado e até a data em que esse trabalho foi escrito, a versão mais atual do Subversion era a 1.6.15, disponível em http://subversion.apache.org.

Este capítulo apresenta um histórico da evolução da ferramenta e o cenário que deu origem ao seu surgimento. São apresentados também os principais comandos e funcionalidades do Subversion.

Uma visão geral da ferramenta também é abordada, visando destacar suas principais características, com o intuito de reunir informações importantes que sirvam de parâmetros para a comparação com outras ferramentas de controle de versões.

As informações contidas neste capítulo foram obtidas em Sussman, Fitzpatrick, Pilato (2009) e através da instalação e uso da versão 1.6.15 do Subversion.

3.1 HISTÓRICO

A principal motivação para o desenvolvimento do Subversion foi o sucesso que o CVS obteve, além do grande número de erros encontrados no mesmo. O desenvolvimento do CVS teve início em 1986 com Dick Grune, porém a ferramenta tinha outro nome e somente entre os anos de 1989 e 1990, Brian Berliner começou o desenvolvimento do CVS que se tornou popular.

Apesar da ótima aceitação do CVS, com o tempo, a grande comunidade de desenvolvedores que participava de seu desenvolvimento acabou encontrando erros considerados críticos, que sugeriam o desenvolvimento de uma nova ferramenta, com o objetivo principal de corrigir os erros cometidos logo no início do projeto. Foi neste cenário, que no começo do ano 2000, a Collabnet, empresa norte americana, começou a recrutar desenvolvedores para trabalhar em um projeto que tinha como foco principal o desenvolvimento do sucessor do CVS, neste caso o Subversion.

O nome Subversion foi originalmente criado por Jim Blandy, que junto com Karl Fogel, Collins-Sussman, Brian Behlendorf e Jason Robbins formaram o primeiro grupo de desenvolvedores do novo projeto. Apesar da idéia principal de desenvolver uma ferramenta

de controle de versões do início, os idealizadores do projeto não queriam que ela fosse totalmente diferente do CVS, e sim, apenas que os erros do CVS fossem corrigidos. Essa estratégia foi adotada visando facilitar a migração dos usuários do CVS para o Subversion, porém ela foi muito criticada por outros desenvolvedores, principalmente aqueles que se envolveram no desenvolvimento de outras ferramentas de controle de versões, como Linus Torvalds, criador do GIT e do kernel do sistema operacional Linux. Segundos Linus, essa estratégia acabou forçando os desenvolvedores envolvidos no projeto a conviver e a aceitar alguns erros do CVS, em prol de facilitar a usabilidade e adaptação ao Subversion pelos antigos usuários do CVS.

Em 2001, o projeto Subversion avançou o bastante para que a ferramenta passasse a controlar o seu próprio código-fonte. Em novembro de 2009, o Subversion foi aceito na Incubadora Apache, marcando o início do processo para se tornar um projeto de padrão alto nível Apache, feito este alcançado em 17 de fevereiro de 2010.

3.2 CICLO BÁSICO DE TRABALHO

Os principais comandos do Subversion estão ligados ao seu ciclo básico de trabalho e fazem parte as seguintes operações:

- **atualizar cópia de trabalho**: através do comando *svn update*, que traz as alterações do repositório para sua cópia de trabalho,
- **realizar alterações:** através dos comandos *svn add*, que adiciona arquivos, diretórios ou links simbólicos, *svn delete*, que exclui um item de uma cópia de trabalho ou do repositório, *svn copy*, que copia um arquivo ou diretório em uma cópia de trabalho ou no repositório e *svn move*, que move um arquivo ou diretório**,**
- **verificar alterações**: através dos comandos *svn status*, que exibe informações sobre o estado de arquivos e diretórios na cópia de trabalho e *svn diff*, que exibe as diferenças entre duas revisões ou caminhos,
- **possivelmente desfazer algumas alterações**: com o comando *svn revert*, que desfaz todas as edições locais,
- **resolver conflitos**: com os comandos *svn update*, que atualiza sua cópia de trabalho e *svn resolved*, que remove arquivos ou diretórios da cópia de trabalho do estado de "conflito" e

- **submeter alterações**: através do comando *svn commit*, que envia as alterações de sua cópia de trabalho para o repositório.

A imagem 3.1 mostra o fluxo de trabalho do Subversion.

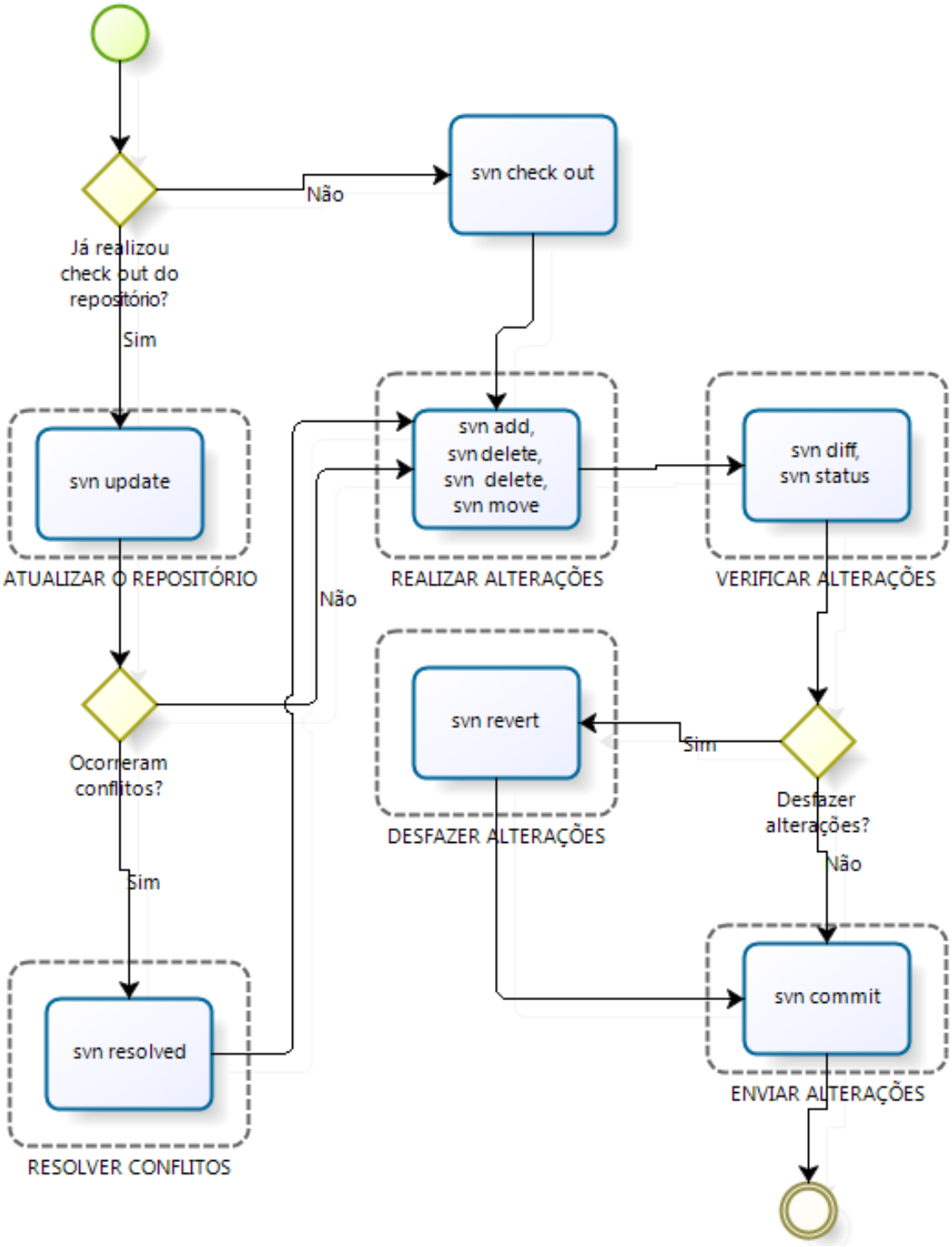


FIGURA 3.1 – Fluxo de trabalho do Subversion

3.3 VISÃO GERAL

3.3.1 LICENÇA

O Subversion seguia os termos da *Debian Free Software Guidelines* (DFSG), que define que qualquer pessoa é livre para baixar o código da ferramenta, modificá-lo, e redistribuí-lo conforme lhe convier; não sendo necessário pedir permissão aos seus criadores. Com o apoio da Apache Foudation, o software passou a adotar os termos da licença Apache.

A licença Apache é uma licença de software livre de autoria da *Apache Software Foundation* (ASF). Ela requer preservação do aviso de direitos autorais e aviso legal, porém, permite o uso do código fonte para o desenvolvimento de software proprietário, bem como software livre e open source.

3.3.2 STATUS TÉCNICO

A portabilidade do Subversion é um ponto forte da ferramenta. Tanto o cliente como o servidor trabalham com os principais sistemas operacionais: Unix, Windows e Mac OS X. Os desenvolvedores do Subversion se preocuparam com questões como a representação de finais de linhas diferentes entre Windows e Linux. Também se preocuparam com o fato de plataformas Unix usarem permissões dos sistemas de arquivo para determinar a executabilidade, enquanto no Windows são usadas as extensões no nome do arquivo. Isso facilita muito o trabalho de desenvolvedores que precisam trabalhar com freqüência em diferentes sistemas operacionais, pois o esforço da equipe de desenvolvimento do Subversion em sanar os problemas causados por essas diferenças, fez com que os comandos em linha de código da ferramenta fossem sempre os mesmos, independente do sistema operacional.

A interoperabilidade do Subversion também é outro ponto forte a favor da ferramenta. Estão disponíveis *plug-ins* para várias IDEs como o Subclipse e o Subversive para o Eclipse e AnkhSVN e VisualSVN para o Visual Studio. O NetBeans já possui funcionalidades nativas que permitem trabalhar com o Subversion.

O Subversion possui uma vasta documentação disponível. Há um livro *on line* gratuito, redigido por alguns de seus principais desenvolvedores, e muitos tutoriais disponíveis na Internet. Este livro foi escrito no formato DocBook/XML, o que possibilita

a conversão do mesmo para vários outros formatos. Além deste livro, e de vários outros de autoria de terceiros, há muita informação sobre o Subversion em *sites*, *blogs* e fóruns especializados na Internet. O cliente de linha de comando do Subversion é auto-documentado e a qualquer momento o usuário pode acionar o comando *svn help* e um subcomando, para descrever a sintaxe, as opções e o comportamento deste subcomando.

A dificuldade de implantação do Subversion é um de seus pontos fracos, comparado a muitas ferramentas de controle de versões mais modernas, pois ele não está disponível em um pacote completo para instalação, o que gera certo trabalho para instalar seus componentes. O processo de instalação do servidor Subversion é realizado através de um arquivo executável de Setup, porém ele possui como dependência a instalação do módulo Apache 2. Para o cliente, o processo é mais simples e existem ferramentas como o TortoiseSvn, de fácil instalação e que exigem poucos parâmetros para o usuário começar a usar o Subversion.

Outra importante característica do Subversion é que com ele, é possível trabalhar com vários protocolos de rede. O Subversion trabalha com protocolos baseados em http e https ou pode trabalhar também com seu próprio protocolo.

3.3.3 OPERAÇÕES DE REPOSITÓRIO

No Subversion a operação de *commit* é atômica, ou seja, se a operação for interrompida pelo meio ela é desconsiderada. Isso permite, por exemplo, que em uma ocasião de queda de energia no exato momento em que era realizada uma operação de *commit*, o repositório não fique um estado inconsistente.

Nas primeiras versões do Subversion não havia a possibilidade de clonar repositórios remotos, de maneira que esses repositórios fossem replicados para uma área local e operassem com as mesmas funcionalidades do repositório remoto. Porém, as versões mais recentes do Subversion possuem uma ferramenta chamada *svnsync* que possibilita essa replicação. Como o Subversion possui uma grande comunidade de colaboradores, não é difícil encontrar componentes desenvolvidos por terceiros que também realizem essa função. São exemplos desses componentes o SVN::Mirror e o SVN-Pusher.

O Subversion é uma ferramenta de controle de versões que trabalha de forma centralizada, por isso não possui a funcionalidade de propagar as mudanças de um

repositório para outro. Essa funcionalidade está presente em ferramentas de controle de versões que trabalham de forma distribuída, utilizando os comandos *push* e *pull*. Porém, os mesmos componentes citados anteriormente, SVN::Mirror e SVN-Pusher, também oferecem essa funcionalidade ao Subversion.

É possível definir permissões de acesso para diferentes partes do repositório remoto no Subversion, diferentemente de muitas outras ferramentas, que somente permitem definição de permissões para o repositório como um todo. Isso se deve ao fato do Subversion operar com permissões através do protocolo http.

O histórico do Subversion além das revisões, quem realizou as alterações e quando elas foram realizadas em um arquivo ou diretório. Através do comando *svn blame* é possível também obter informações de autor e revisões por linha de um arquivo ou URL especificados.

A cada *commit* realizado no Subversion é criado um *changeset* diferente com informações como a lista de arquivos alterados, quem realizou as alterações, mensagem de *log* e quando foi realizado. Porém, o Subversion não permite que o próprio usuário crie um *changeset* da maneira que desejar, como em outras ferramentas de controle de versões.

No caso de um *merge* com um arquivo renomeado, o Subversion não consegue reconhecer o arquivo e copia o arquivo de origem para o local destinado, deletando o arquivo de destino.

Movimentar e renomear arquivos ou diretórios no Subversion são operações possíveis e simples. As cópias de arquivos e diretórios também são suportadas. Apesar de parecer uma operação simples, muitas ferramentas de controle de versão não suportam a operação de cópia. Muitos usuários também utilizam a possibilidade de cópia de arquivos e diretórios para criarem ramos manualmente no Subversion.

3.3.4 CARACTERÍSTICAS

Uma importante característica do Subversion é que ele consegue controlar as alterações realizadas na cópia de trabalho que ainda não foram enviadas ao repositório. Isso é feito através do comando *svn diff*, assim o usuário pode analisar suas alterações com cautela antes de enviá-las ao repositório.

Todo *commit* no Subversion pode ser acompanhado de uma mensagem de *log*, ou seja, o usuário pode descrever todas as alterações relativas àquele *commit* em uma

mensagem, que fica gravada no repositório. O Subversion permite apenas uma mensagem de *log* para todo o conjunto de mudanças, não permitindo que cada arquivo possua uma mensagem de *log* particular.

O Subversion oferece ao usuário a opção de não somente realizar um *check out* no repositório como um todo, mas também um *check out* parcial, ou seja, o usuário tem a opção de realizar um *check out* em apenas um diretório por exemplo. Assim como *check out*, o Subversion também permite o *commit* parcial, ou seja, o usuário pode realizar o *commit* selecionando apenas o diretório desejado, por exemplo.

3.3.5 INTERFACE

O Subversion é uma das ferramentas de controle de versões com o maior número de clientes baseados em interface *Web*. Geralmente, esses clientes possibilitam que o usuário navegue através da árvore e das várias revisões dos arquivos, além de possibilitar a execução de *diffs*, o trabalho com o formato rss, a execução de *check outs*, entre outras funcionalidades. São exemplos de clientes de interface web para o Subversion as ferramentas ViewVC, SVN::Web, WebSVN, ViewSVN, mod_svn_view, Chora, Trac (Figura 3.2), SVN::RaWeb::Light, SVN Browser, Insurrection e perl_svn.

FIGURA 3.2 – Trac Subversion

Existem também muitos clientes que trabalham em modo gráfico, ou GUIs (*Graphical User Interfaces*), para o Subversion. Esta é uma característica importante na avaliação de ferramentas de controle de versões, pois muito desenvolvedores não gostam de trabalhar com linhas de comando. São exemplos de GUIs para o Subversion as ferramentas RapidSVN, que é multiplataforma, TortoiseSVN (Figura 3.3), que funciona

como um *plug in* para o Windows Explorer e Jsvn, desenvolvido com a API Java para interfaces gráficas Swing.

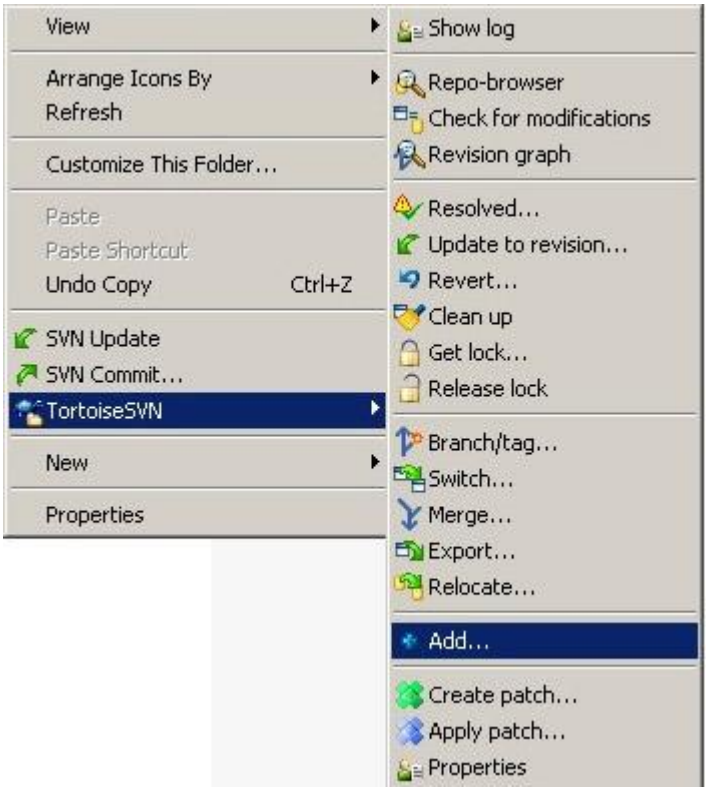


FIGURA 3.3 – Tortoisesvn

3.4 CONCLUSÃO

O Subversion conseguiu atender os objetivos principais da sua equipe de projetistas; ser uma ferramenta que substituísse o CVS e com soluções para os erros encontrados no passado.

Apesar de ser uma ferramenta de controle de versões que trabalha de forma centralizada, o Subversion se beneficia da grande comunidade de desenvolvedores que possui. Isso faz com que suas funcionalidades sejam estendidas ao ponto de realizarem operações que, teoricamente, somente seriam viáveis em ferramentas de controle de versões que trabalham de forma distribuída.

Mesmo com os avanços significativos obtidos pelo Subversion, o forte crescimento da Internet e dos ambientes de colaboração exigiram ferramentas que atendessem as demandas de grandes comunidades de desenvolvedores espalhadas pelo mundo. O modelo

de ferramentas de controle de versões baseadas na arquitetura cliente/servidor já não estavam atendendo bem essas comunidades, pois exigiam uma disponibilidade grande de banda e uma capacidade enorme de processamento do servidor.

A grande documentação disponível para o Subversion é fruto também de sua grande comunidade de usuários e desenvolvedores. Apesar de seu processo de instalação não ser trivial como de outras ferramentas semelhantes, é fácil encontrar tutoriais de instalação em livros e na Internet.

4 GIT

Neste capítulo é apresentada a segunda das três ferramentas de controle de versões comparadas, o GIT, que é um SCV descentralizado e até a data em que esse trabalho foi escrito, a última versão da ferramenta era a 1.7.3.2, disponível em http://git-scm.com/download.

O capítulo apresenta um histórico da evolução da ferramenta e o cenário que deu origem ao seu surgimento. São apresentados também os principais comandos e funcionalidades do GIT.

Uma visão geral da ferramenta também é abordada, visando destacar suas principais características, com o intuito de reunir informações importantes que sirvam de parâmetros para a comparação com outras ferramentas de controle de versões.

As informações contidas neste capítulo foram obtidas em Chacon (2009) e através da instalação e uso da versão 1.7.3.2 do GIT.

4.1 HISTÓRICO

O desenvolvimento do GIT teve início quando vários desenvolvedores do Kernel Linux resolveram deixar de usar o BitKeeper. O BitKeeper é um sistema de controle de versões descentralizado e que possuia licença gratuita, porém, Larry McVoy, detentor dos direitos autorais da ferramenta, optou por torná-la proprietária e o BitKeeper passou a ser pago. A filosofia de software livre adotado pelo Linux não era compatível com a própria ferramenta usada para controlar a evolução do código de seu kernel, o que levou Linus Torvalds a procurar outra ferramenta para tal. Linus e a comunidade de desenvolvedores do Linux não ficaram satisfeitos com nenhuma das ferramentas que trabalhavam de forma descentralizada disponíveis naquela época, e optaram então por desenvolver a sua própria ferramenta de controle de versões.

Linus dizia que seria importante que essa nova ferramenta não trabalhasse de forma semelhante ao CVS, segundo ele, um exemplo de software mal projetado. Linus afirmou que o Subversion já era um exemplo do que não fazer, pois seu *slogan*, “Um CVS feito da maneira correta”, já demonstrava o fracasso do projeto. Linus disse que era impossível desenvolver um CVS de forma correta.

Outro ponto importante que deu origem às características de projeto do GIT, é que Linus afirmou que a ferramenta deveria ser semelhante ao BitKeeper, no sentido de trabalhar de forma distribuída. Alto desempenho e medidas de segurança contra corrompimento de arquivos também eram necessários.

Linus comandou o desenvolvimento do GIT até ele começar a ser utilizado por outros usuários com conhecimentos técnicos avançados. Neste momento, Junio Hamano assumiu a gerência do projeto e foi responsável pelo lançamento da versão 1.0 em 21 de dezembro de 2005 e ainda era o responsável principal pelo desenvolvimento do GIT quando essa monografia foi escrita.

4.2 CICLO BÁSICO DE TRABALHO

Os principais comandos do GIT estão ligados ao seu ciclo básico de trabalho. O ciclo básico de trabalho pode ser definido como as operações comuns realizadas por um desenvolvedor em seu dia a dia de trabalho. Fazem parte do ciclo básico do GIT as seguintes operações:

- **atualizar seu repositório e cópia de trabalho**: através do comando *git fetch*, que faz o *download* de alterações do repositório desejado, seguido de um comando *git merge*, para mesclar as alterações baixadas com o seu repositório, ou através do comando *git pull*, que atualiza seu repositório e sua área de trabalho com as alterações de outro repositório,
- **fazer alterações:** através dos comandos *git add*, que adiciona arquivos ao index, *git rm*, que exclui um item de uma cópia de trabalho ou do repositório e *git mv*, que move ou renomeia um arquivo ou diretório**,**
- **verificar suas alterações**: através dos comandos *git status*, que exibe informações sobre o estado de arquivos e diretórios na cópia de trabalho e *git diff*, que exibe as diferenças entre duas revisões ou caminhos,
- **possivelmente desfazer algumas alterações**: com o comando *git revert*, que desfaz todas as edições locais,
- **resolver conflitos**: com o comando *git mergetool*, que executa ferramentas de resolução de conflitos para resolver conflitos de junção,
- **submeter alterações**: através do comando *git commit*, que envia as alterações de sua cópia de trabalho para o repositório e

- **possivelmente propagar suas alterações para outro repositório**: através do comando *git push*, que envia as alterações do repositório local (origem) para outro repositório (destino).

A figura 4.1 demonstra o fluxo de trabalho do Git.

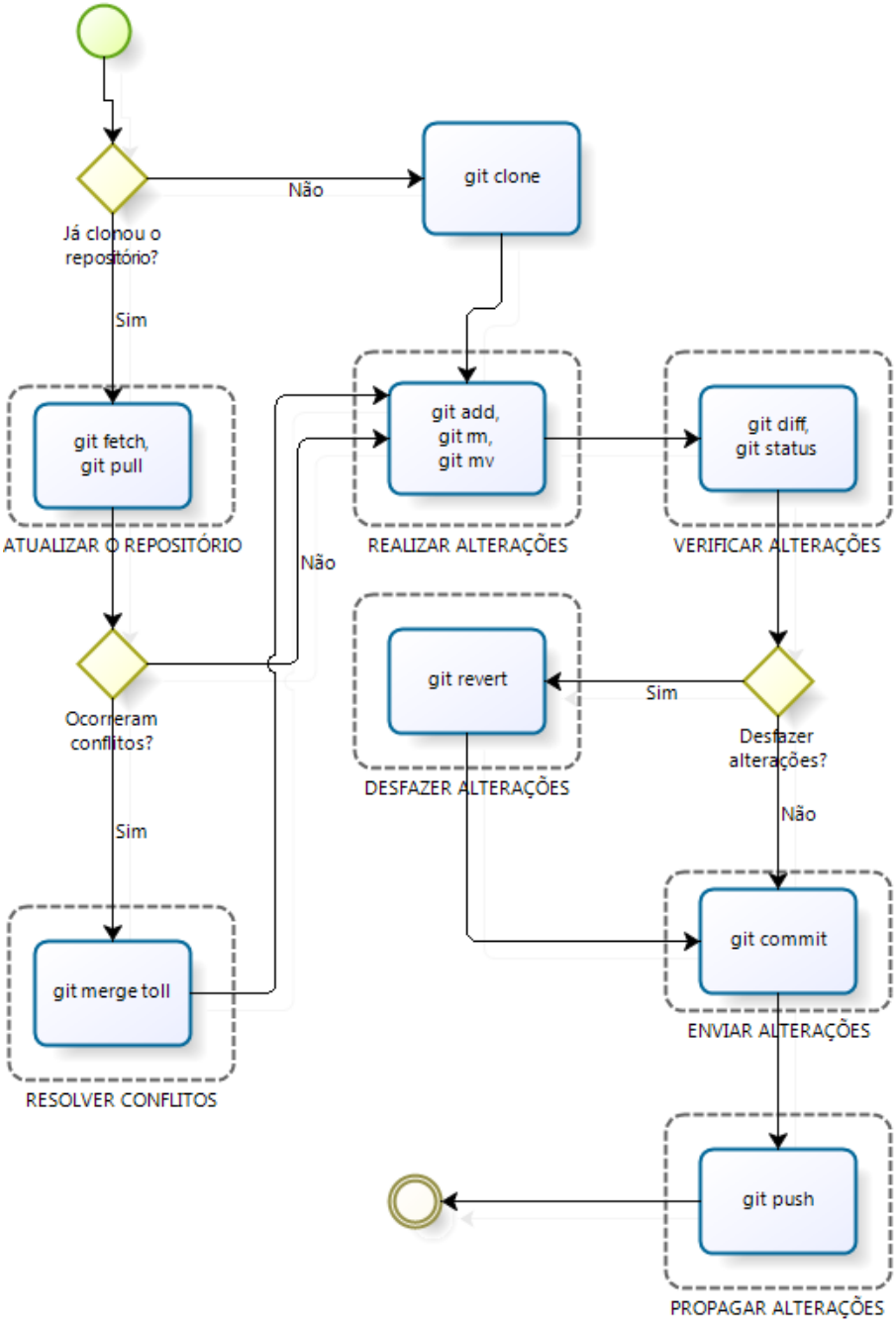


FIGURA 4.1 – Fluxo de trabalho do Git

4.3 VISÃO GERAL

4.3.1 LICENÇA

O GIT segue os termos da licença General Public License (GPL). A GPL foi criada por Richard Stallman em 27 de setembro de 1983 para o projeto GNU, um projeto de software livre. Até a data em que essa monografia foi escrita, existiam três versões para a licença GPL. O GIT segue os termos da segunda versão.

De acordo com Richard Stallman a principal mudança realizada na segunda versão da licença GPL é a inclusão de uma cláusula chamada por ele de “Liberdade ou Morte”. Essa cláusula define que ninguém pode impor restrições a terceiros de distribuir seu software, restringindo a distribuição em forma binária, por exemplo (STALLMAN, 2006).

4.3.2 STATUS TÉCNICO

Como o GIT foi inicialmente desenvolvido por Linus, com o intuito de criar uma ferramenta para controlar o desenvolvimento do kernel do Linux, o foco do seu projeto foi criar uma ferramenta de controle de versões desenvolvida para usuários do Linux e que tivesse bom desempenho neste sistema operacional. Apesar disso, o GIT também pode ser usado em outros sistemas operacionais como BSD, Solaris e Darwin, baseados na plataforma Unix, e no Windows.

Assim como no Subversion, a interoperabilidade do Git é um ponto forte da ferramenta. Estão disponíveis *plug-ins* para várias IDEs como o JGit e o EGit para o Eclipse, Git Extensions para o Visual Studio e NbGit para o NetBeans.

O GIT possui uma boa documentação disponível. Há vários livros disponíveis, inclusive *on line*. Também são encontrados vários tutoriais, *blogs* e fóruns falando sobre a ferramenta. O próprio *site* oficial do GIT possui uma seção de documentação onde é possível encontrar uma referência com os principais comandos da ferramenta, um tutorial oficial com um ciclo básico de comandos passo a passo, um curso rápido de como utilizar o GIT com o Subversion, um documento que ensina boas práticas a serem adotadas no uso da ferramenta e vários outros tipos de documentos. Para reunir toda essa informação também foi criado um wiki que reúne apenas links e uma breve descrição sobre vários tipos de documentos da ferramenta. Também é possível usar o comando *git help* e um subcomando para descrever a sintaxe, as opções e o comportamento deste subcomando.

A grande vantagem da implantação do GIT é o fato de existirem pacotes completos para a sua instalação. No Linux a instalação pode ser feita através de seu sistema nativo de

gerenciamento de pacotes, através dos comandos $ *yum install git-core* e *$ apt-get install git-core* ou através do download dos pacotes no formato .deb ou .rpm. Para o Windows esta disponível o pacote msysgit, que possui diferentes versões, desde um instalador portátil até um instalador com todo o código fonte do GIT e um compilador em C. A instalação em plataformas Unix e em ambiente Mac também é trivial.

Outra importante característica do GIT é que com ele, é possível trabalhar com vários protocolos de rede. O GIT trabalha com os protocolos rsync, ssh, http, https ou pode trabalhar também com seu próprio protocolo. O protocolo git procura otimizar a largura de banda utilizada e por isso torna operações de atualizações rápidas e eficientes.

4.3.3 OPERAÇÕES DE REPOSITÓRIO

Assim como no Subversion, no GIT a operação de *commit* é atômica, ou seja, se uma operação é interrompida, ela é desconsiderada e o repositório não fica em um estado inconsistente.

Pelo fato de ser um sistema de controle de versões descentralizado, clonar um repositório no GIT é uma operação comum e trivial. Isso é feito através do comando *git clone*. Este comando clona um repositório em um novo diretório, cria ramos remotos para cada ramo do repositório clonado e cria um ramo inicial já realizando um *check out* no mesmo, ou seja, selecionando esse ramo como o ramo ativo. Este ramo inicial criado no novo repositório é um filho do ramo ativo do repositório clonado.

No GIT a propagação de mudanças de um repositório para outro é oferecida através dos comandos *git push*, *git fetch* e *git pull*. O comando *git fetch* atualiza o repositório sem atualizar a cópia de trabalho, o comando *git push* envia as alterações de um repositório local para um repositório remoto e o comando *git pull* atualiza o repositório local com alterações de um repositório remoto. O comando *git pull* já atualiza também a cópia de trabalho do usuário, portanto, ele nada mais é do que um *git fetch* seguido de um *merge*. A figura 4.2 mostra uma sequência de eventos onde dois usuários propagam mudanças entre seus repositórios e um repositório central utilizando o GIT.

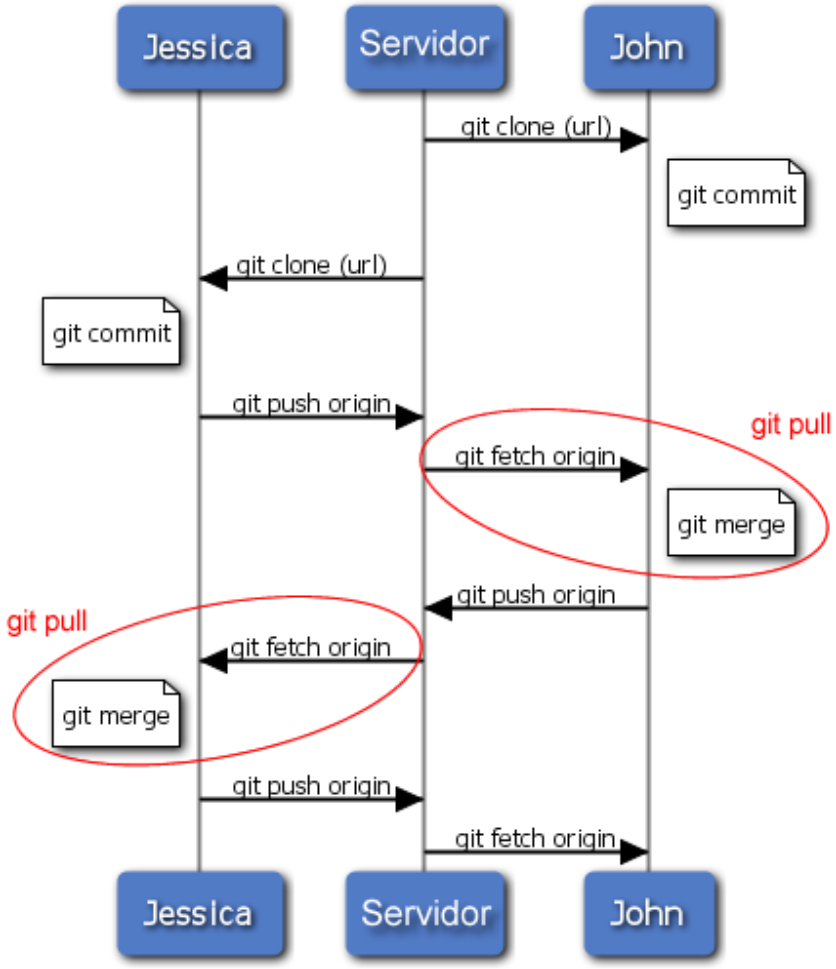


FIGURA 4.2 – Propagação de alterações entre repositórios no Git

O GIT permite definir permissões tanto para o repositório como um todo, quanto para diretórios e arquivos. O controle dessas permissões pode ser feito de várias maneiras, como por exemplo, através do protocolo ssh, que permite criar chaves para controle de autenticação ou através do procotolo http, que permite definir permissões de leitura e escrita para cada arquivo ou diretório.

Assim como o Subversion, o histórico do GIT registra as revisões, quem realizou as alterações e quandos essas alterações foram realizadas em um arquivo ou diretório. Além disso, é possível analisar as alterações realizadas em um arquivo linha por linha através do comando *git blame*.

Os desenvolvedores do GIT consideram a palavra *changeset* inadequada para a ferramenta, pois alegam que o GIT não grava alterações, e sim estados. Porém, na prática, é possível trabalhar de forma semelhante a outras ferramentas com *changesets* no GIT, através do comando *git log*.

Com o GIT é possível mover arquivos e diretórios, porém ele não oferece suporte à operação de *rename* como acontece no Subversion. Na verdade, essa funcionalidade esta disponível através do comando *git mv*, que ao invés de renomear o arquivo, o apaga, fazendo em seguida uma réplica idêntica à apagada, com o nome desejado.

No caso de um *merge* com um arquivo renomeado, o comportamento do GIT também é semelhante ao do Subversion, ou seja, ele não consegue reconhecer o arquivo e copia o arquivo de origem para o local destinado, apagando o arquivo de destino. A operação de cópia não é suportada no GIT.

4.3.4 CARACTERÍSTICAS

Com o GIT é possível controlar as alterações realizadas na cópia de trabalho que ainda não foram enviadas ao repositório. Isso é feito através do comando *git diff*, assim o usuário pode analisar suas alterações com cautela antes de enviá-las ao repositório. É possível também ignorar alterações indesejadas, alocando-as em um diretório isolado, através do comando *git stash*.

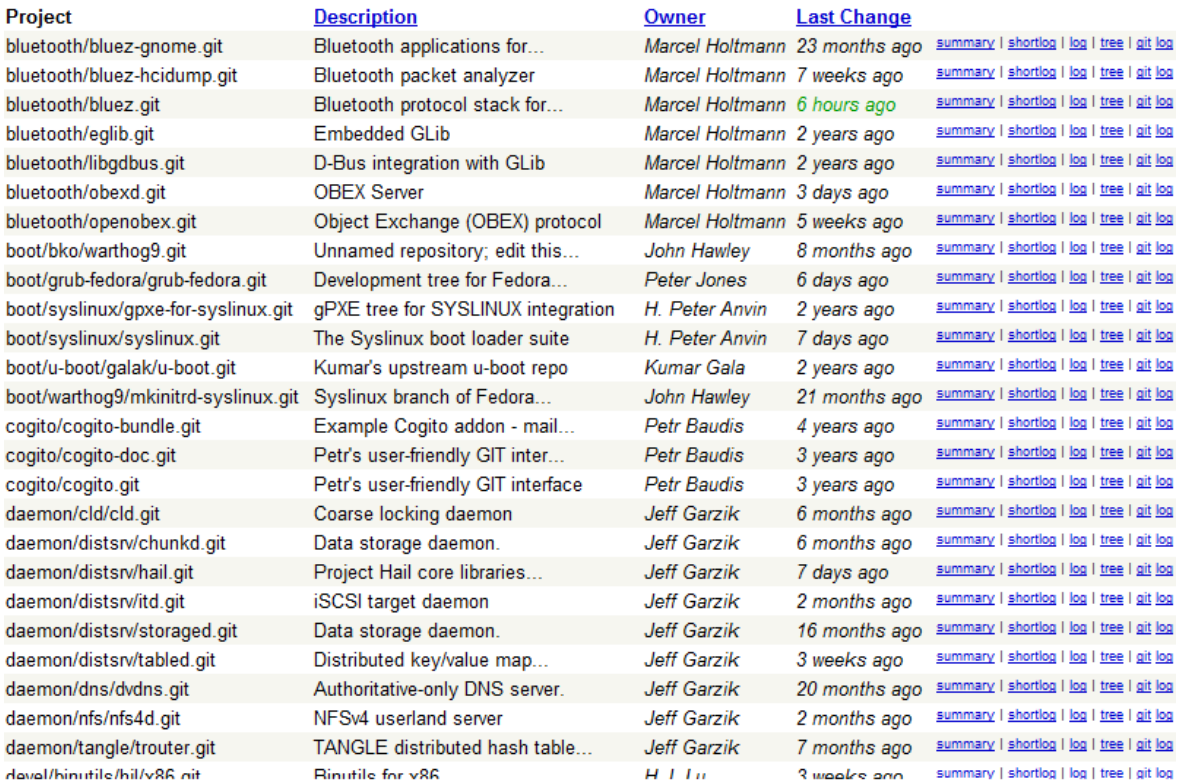
Como no Subversion, no GIT todo *commit* pode ser acompanhado de uma mensagem de *log*, ou seja, o usuário pode descrever todas as alterações relativas aquele *commit* em uma mensagem, que fica gravada no repositório. O GIT também permite apenas uma mensagem de *log* para todo o conjunto de mudanças, não permitindo que cada arquivo do conjunto de mudanças possua uma mensagem de *log* particular.

Diferentemente do Subversion, o GIT não oferece ao usuário a opção de realizar um *check out* parcial, ou seja, realizar um *check out* em apenas um diretório por exemplo.

4.3.5 INTERFACE

O GIT não possui tantos clientes baseados em interface *Web* como o Subversion, porém sua distribuição já possui um cliente, o *Gitweb*. O *Gitweb* foi desenvolvido na linguagem Perl e permite navegar pelos diretórios através de revisões desejadas, ver o conteúdo dos arquivos, verificar o *log* de diferentes ramos e examinar os *commits*, visualizando a mensagem de *log* e as alterações relativas ao *commit*. Além disso, ele trabalha com *feeds* em rss ou no formato atom. Outro bom cliente de interface web para o GIT é o Cgit, que

possui uma interface melhor do que a do gitweb e disponibiliza ainda mais detalhes. A figura 4.3 demonstra a interface do cliente gitweb.



Project	Description	Owner	Last Change	
bluetooth/bluez-gnome.git	Bluetooth applications for...	Marcel Holtmann	23 months ago	summary | shortlog | log | tree | git log
bluetooth/bluez-hcidump.git	Bluetooth packet analyzer	Marcel Holtmann	7 weeks ago	summary | shortlog | log | tree | git log
bluetooth/bluez.git	Bluetooth protocol stack for...	Marcel Holtmann	6 hours ago	summary | shortlog | log | tree | git log
bluetooth/eglib.git	Embedded GLib	Marcel Holtmann	2 years ago	summary | shortlog | log | tree | git log
bluetooth/libgdbus.git	D-Bus integration with GLib	Marcel Holtmann	2 years ago	summary | shortlog | log | tree | git log
bluetooth/obexd.git	OBEX Server	Marcel Holtmann	3 days ago	summary | shortlog | log | tree | git log
bluetooth/openobex.git	Object Exchange (OBEX) protocol	Marcel Holtmann	5 weeks ago	summary | shortlog | log | tree | git log
boot/bko/warthog9.git	Unnamed repository; edit this...	John Hawley	8 months ago	summary | shortlog | log | tree | git log
boot/grub-fedora/grub-fedora.git	Development tree for Fedora...	Peter Jones	6 days ago	summary | shortlog | log | tree | git log
boot/syslinux/gpxe-for-syslinux.git	gPXE tree for SYSLINUX integration	H. Peter Anvin	2 years ago	summary | shortlog | log | tree | git log
boot/syslinux/syslinux.git	The Syslinux boot loader suite	H. Peter Anvin	7 days ago	summary | shortlog | log | tree | git log
boot/u-boot/galak/u-boot.git	Kumar's upstream u-boot repo	Kumar Gala	2 years ago	summary | shortlog | log | tree | git log
boot/warthog9/mkinitrd-syslinux.git	Syslinux branch of Fedora...	John Hawley	21 months ago	summary | shortlog | log | tree | git log
cogito/cogito-bundle.git	Example Cogito addon - mail...	Petr Baudis	4 years ago	summary | shortlog | log | tree | git log
cogito/cogito-doc.git	Petr's user-friendly GIT inter...	Petr Baudis	3 years ago	summary | shortlog | log | tree | git log
cogito/cogito.git	Petr's user-friendly GIT interface	Petr Baudis	3 years ago	summary | shortlog | log | tree | git log
daemon/cld/cld.git	Coarse locking daemon	Jeff Garzik	6 months ago	summary | shortlog | log | tree | git log
daemon/distsrv/chunkd.git	Data storage daemon.	Jeff Garzik	6 months ago	summary | shortlog | log | tree | git log
daemon/distsrv/hail.git	Project Hail core libraries...	Jeff Garzik	7 days ago	summary | shortlog | log | tree | git log
daemon/distsrv/itd.git	iSCSI target daemon	Jeff Garzik	2 months ago	summary | shortlog | log | tree | git log
daemon/distsrv/storaged.git	Data storage daemon.	Jeff Garzik	16 months ago	summary | shortlog | log | tree | git log
daemon/distsrv/tabled.git	Distributed key/value map...	Jeff Garzik	3 weeks ago	summary | shortlog | log | tree | git log
daemon/dns/dvdns.git	Authoritative-only DNS server.	Jeff Garzik	20 months ago	summary | shortlog | log | tree | git log
daemon/nfs/nfs4d.git	NFSv4 userland server	Jeff Garzik	2 months ago	summary | shortlog | log | tree | git log
daemon/tangle/trouter.git	TANGLE distributed hash table...	Jeff Garzik	7 months ago	summary | shortlog | log | tree | git log
devel/binutils/hjl/x86.git	Binutils for x86	H. J. Lu	3 weeks ago	summary | shortlog | log | tree | git log

FIGURA 4.3 – Interface do cliente gitweb

Apesar de o GIT ter sido desenvolvido visando um bom desempenho para o sistema operacional Linux, e muitos usuários do Linux preferirem usá-lo através de linha de comando, existem bons clientes que trabalham em modo gráfico, ou *GUIs* , disponíveis para a ferramenta. O cliente Gitk já acompanha a distribuição do GIT. Estão disponíveis também, bons clientes como o Qgit, Git-gui e o TortoiseGit, que foi desenvolvido para usuários do Windows. A figura 4.4 demonstra a interface do cliente Tortoisegit.

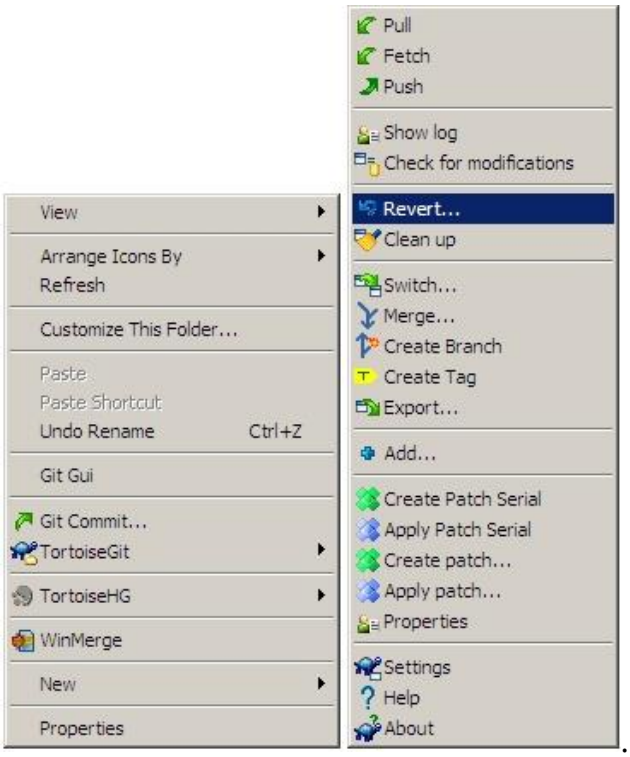


FIGURA 4.4 – Interface do cliente TortoiseGit

4.4 CONCLUSÃO

O GIT conseguiu atender os objetivos principais que Linus Torvalds e a comunidade de usuários do Linux queriam, uma ferramenta de controle de versões que substituísse a função do Bitkeeper de controlar as versões do kernel do Linux, além de ter alto desempenho rodando neste sistema operacional.

O envolvimento de Linus Torvalds no projeto e o apoio da comunidade de usuários do Linux foi também fundamental para o sucesso do GIT. Como a ferramenta foi desenvolvida alguns anos depois de projetos conhecidos como Subversion e BitKeeper, os desenvolvedores já conheciam uma série de problemas operacionais e de desempenho e puderam estudá-los com o intuito de não cometer os mesmos erros no desenvolvimento do GIT.

Mesmo com uma boa documentação disponível, a curva de aprendizagem do GIT é provavelmente a mais íngreme entre as ferramentas comparadas, por isso ele é recomendado para usuários que já possuam algum grau de conhecimento em controle de versões.

5 MERCURIAL

Neste capítulo é apresentada a terceira das ferramentas de controle de versões comparadas, o Mercurial. O Mercurial é um SCV descentralizado e até a data em que esse trabalho foi escrito, a última versão disponível da ferramenta era a 1.7.2, disponível em http://mercurial.selenic.com/downloads/.

O capítulo apresenta um histórico da evolução da ferramenta e o cenário que deu origem ao seu surgimento. São apresentados também os principais comandos e funcionalidades do Mercurial.

Uma visão geral da ferramenta também é abordada, visando destacar suas principais características, com o intuito de reunir informações importantes que sirvam de parâmetros para a comparação com outras ferramentas de controle de versões.

As informações contidas neste capítulo foram obtidas em O'sullivan (2009) e através da instalação e uso da versão 1.7.2 do Mercurial.

5.1 HISTÓRICO

Assim como o GIT, o desenvolvimento do Mercurial teve início quando vários desenvolvedores do kernel do Linux resolveram deixar de usar o BitKeeper, pois Larry MacVoy, detentor dos direitos autorais da ferramenta, optou por torná-la proprietária e o BitKeeper passou a ser pago.

Após o anúncio de Larry MacVoy, Linus Torvalds, criador do kernel do Linux, anunciou que estava a procura de uma nova ferramenta para substituir o BitKeeper. Matt Mackall, o criador e líder de desenvolvimento do Mercurial, começou então a trabalhar no desenvolvimento de seu sistema de controle de versões. Paralelamente, Linus anunciou que também começara a desenvolver um novo sistema de controle de versões, o GIT.

O release 0.1 do Mercurial foi anunciando por Matt Mackall no dia 19 de abril de 2005. Apesar de já possuir boas características como uma interface funcional e um armazenamento eficiente, o Mercurial não foi adotado como ferramenta para controlar o código do kernel do Linux. A ferramenta escolhida foi o GIT.

Apesar de não ter sido escolhido, o Mercurial acabou sendo adotado em muitos projetos importantes e passou a ser, assim como o GIT, um dos SCVs descentralizados mais populares.

5.2 CICLO BÁSICO DE TRABALHO

Os principais comandos do Mercurial estão ligados ao seu ciclo básico de trabalho que pode ser definido como as operações comuns realizadas por um desenvolvedor em seu dia a dia de trabalho. Fazem parte do ciclo básico do Mercurial as seguintes operações ilustrados na Figura 5.1:

- **atualizar repositório e cópia de trabalho**: através do comando *hg pull*, que faz o *download* de alterações do repositório desejado, seguido de um comando *hg update*, para mesclar as alterações baixadas com a sua cópia de trabalho,
- **fazer alterações:** através dos comandos *hg add*, que adiciona arquivos a lista de arquivos controlados, *hg remove*, que exclui um item de uma cópia de trabalho, e *hg mv* ou *hg rename*, que move ou renomeia um arquivo ou diretório**,**
- **verificar alterações**: através dos comandos *hg status*, que exibe informações sobre o estado de arquivos e diretórios na cópia de trabalho e *hg diff*, que exibe as diferenças entre duas revisões ou caminhos,
- **possivelmente desfazer algumas alterações**: com o comando *hg revert*, que desfaz todas as edições locais,
- **resolver conflitos**: com o comando *hg resolve*, que executa ferramentas de resolução de conflitos de junção,
- **submeter alterações**: através do comando *hg commit*, que envia as alterações de sua cópia de trabalho para o repositório e
- **possivelmente propagar suas alterações para outro repositório**: através do comando *hg push*, que envia as alterações do repositório local (origem) para outro repositório (destino).

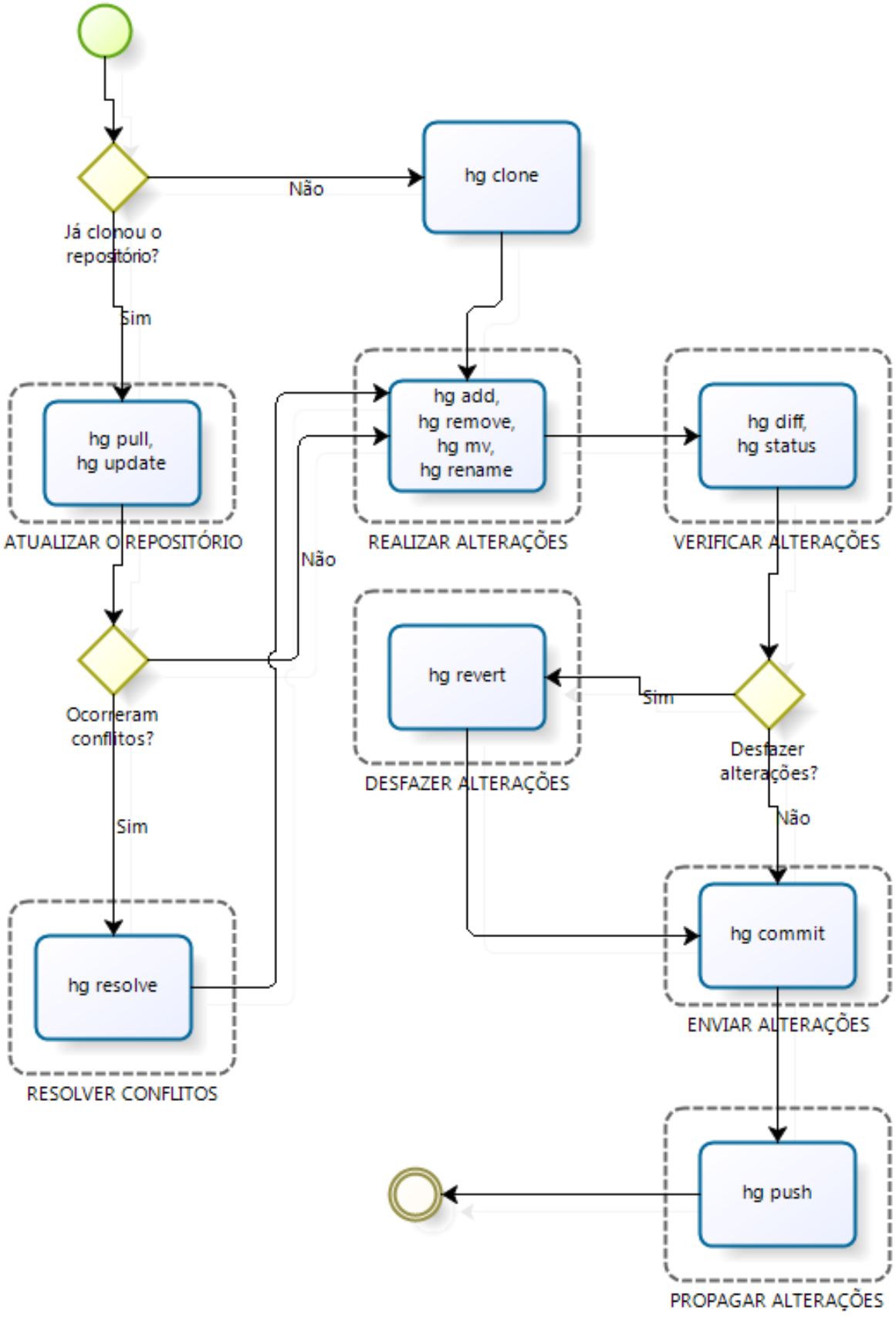


FIGURA 5.1 – Fluxo de trabalho do Mercurial

5.3 VISÃO GERAL

5.3.1 LICENÇA

O Mercurial, assim como o GIT, também segue os termos da segunda versão da licença GPL. Maiores informações sobre essa licença podem ser consultadas na seção 4.3.1.

5.3.2 STATUS TÉCNICO

Apesar de ter sido desenvolvido inicialmente com o intuito de substituir o BitKeeper, no papel de SCV do kernel do Linux, o desenvolvimento do Mercurial não foi tão focado no Linux e ao longo do tempo ele acabou obtendo um melhor desempenho do que o GIT em outros sistemas operacionais. O Mercurial foi desenvolvido na linguagem Python e funciona em qualquer sistema operacional que seja compatível com essa linguagem, como Windows, Linux, Mac OS X e outros.

Assim como no Subversion e no Git, a interoperabilidade é um ponto forte do Mercurial. Estão disponíveis *plug-ins* para várias IDEs como o Mercurial Eclipse para o Eclipse, e *plug-ins* para o Visual Studio e o NetBeans.

O Mercurial possui uma boa documentação disponível. A principal referência da ferramenta é seu manual oficial, chamado *Mercurial: The Definitive Guide*. Esse manual foi escrito por Bryan O'Sullivan, um dos principais colaboradores no desenvolvimento do Mercurial, e pode ser baixado gratuitamente no *site* oficial da ferramenta. No mesmo *site* estão disponíveis tutoriais para iniciantes e documentos que mostram de maneira ilustrada como o Mercurial funciona. Assim como o Subversion e o GIT, o Mercurial também é auto documentado e através do comando *hg help*, e um subcomando, é possível descrever a sintaxe, as opções e o comportamento deste subcomando.

A implantação do Mercurial, assim como a do GIT, leva vantagem sobre a implantação do Subversion, pelo fato de também possuir um pacote completo de instalação. No Windows, a instalação recomendada é através do TortoiseHg, que já instala tanto o cliente de interface gráfica quanto o Mercurial. Para o Linux existem vários pacotes de instalação, alguns específicos para determinadas distribuições como Debian, Ubuntu, Slackware e outras. O site oficial do Mercurial disponibiliza um *Wiki* com informações e

links para *download* de instaladores para vários outros sistemas operacionais como Mac OS X, Solaris, AIX e FreeBSD.

O Mercurial oferece um bom suporte a protocolos de rede e trabalha com os protocolos http e ssh.

5.3.3 OPERAÇÕES DE REPOSITÓRIO

Assim como no Subversion e no GIT, no Mercurial a operação de *commit* é atômica, ou seja, se a operação for interrompida, ela é desconsiderada e o repositório não fica em um estado inconsistente.

GIT e Mercurial são sistemas de controle descentralizados, por isso, clonar um repositório nessas ferramentas é uma operação simples, e muitas vezes rotineira. No Mercurial é possível clonar um repositório através do comando *hg clone*. Esse comando realiza operações semelhantes às operações explicadas para o comando *git clone*, na seção 4.3.3.

No Mercurial também estão presentes comandos para propagar alterações entre repositórios. Esses comandos são o *hg push* e o *hg pull*. O comando *hg push* envia as alterações do repositório local para um repositório remoto, assim como no GIT, porém, o comando *hg pull* realiza operações diferentes do comando *git pull*. O comando *git pull* faz *download* de alterações de outro repositório e realiza um *merge* com a cópia de trabalho, enquanto o comando *hg pull* realiza apenas a etapa de download das alterações. No Mercurial, para realizar o *download* das alterações de outro repositório e um *merge* com a cópia de trabalho, deve-se executar um comando *hg pull* seguido de um comando *hg update*. A figura 5.2 mostra uma sequência de eventos onde dois usuários propagam mudanças entre seus repositórios e um repositório central utilizando o Mercurial.

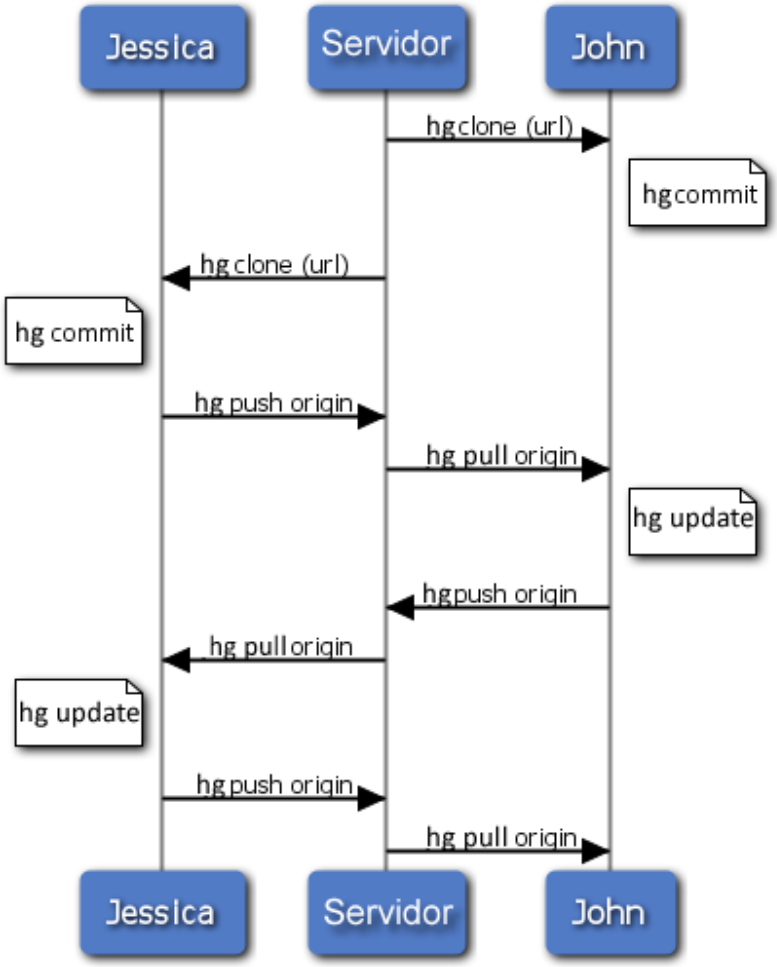


FIGURA 5.2 – Propagação de alterações entre repositórios no Mercurial

O Mercurial também permite definir permissões tanto para o repositório como um todo, quanto para diretórios e arquivos. O controle dessas permissões pode ser feito através dos protocolos ssh e http.

Assim como no Subversion e no GIT, o histórico do Mercurial também registra as revisões, quem realizou as alterações e quandos essas alterações foram realizadas em um arquivo ou diretório. Além disso, é possível analisar as alterações realizadas em um arquivo linha por linha através do comando *hg annotate*.

No Mercurial cada *changeset* possui um número de identificação chamado *changeset ID*. O Mercurial grava todos os *changesets* do repositório em um arquivo chamado *changelog*.

Com o Mercurial é possível mover arquivos e diretórios, porém ele não oferece suporte à operação de *rename* como acontece no Subversion. Essa funcionalidade esta disponível através do comando *hg rename* ou *hg mv*, que na verdade são comandos

idênticos com nomes diferentes. Como acontece no GIT, ao invés de renomear o arquivo, o Mercurial o apaga, fazendo em seguida uma réplica idêntica à apagada, com o nome desejado.

O Mercurial, diferentemente do Subversion e do GIT, trabalha com *merge* inteligente após operações de *rename* em um arquivo. Isso significa que se um arquivo é alterado e depois renomeado com um nome diferente, e em seguida, é feito um *merge* entre o arquivo com nome antigo e o com nome novo, o Mercurial consegue identificar que os dois arquivos são os mesmos e realizar o *merge* corretamente. Assim como no Subversion e diferentemente do GIT, as operações de cópia são suportadas no Mercurial.

5.3.4 CARACTERÍSTICAS

Assim como no GIT e no Subversion, no Mercurial é possível controlar as alterações realizadas na cópia de trabalho e que ainda não foram enviadas ao repositório. Isso é feito através do comando *hg diff*, e o usuário pode analisar suas alterações com cautela antes de enviá-las ao repositório.

Também como no Subversion e no GIT, no Mercurial todo *commit* pode ser acompanhado de uma mensagem de *log*, ou seja, o usuário pode descrever todas as alterações relativas aquele *commit* em uma mensagem, que fica gravada no repositório. O Mercurial também permite apenas uma mensagem de *log* para todo o conjunto de mudanças, não permitindo que cada arquivo possua uma mensagem de *log* particular.

Diferentemente do Subversion, e assim como o GIT, o Mercurial não oferece ao usuário a opção de realizar um *check out* parcial, ou seja, em apenas um diretório por exemplo. A operação de *commit* pode ser restringida, usando o gerenciamento de permissões.

5.3.5 INTERFACE

No pacote de instalação do Mercurial já está incluso um cliente baseado em interface *Web*. Porém, o Mercurial possui outras ferramentas semelhantes e com funcionalidades melhores. O *plug in* TracMercurial permite utilizar o Trac, uma ferramenta de controle de mudanças, como um navegador para o repositório do Mercurial (Figura 5.3).

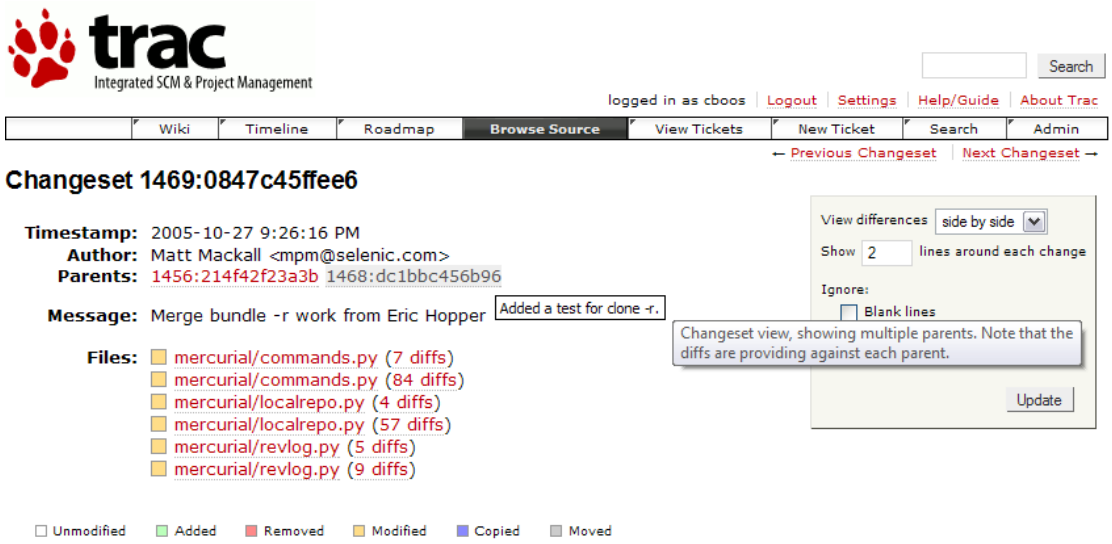


FIGURA 5.3 – Trac Mercurial

O Mercurial possui muitos clientes que trabalham em modo gráfico. No *site* oficial do Mercurial os desenvolvedores mantêm uma lista com links e uma breve explicação de cada ferramenta. Assim como estão disponíveis o TortoiseSvn para o Subversion e o TortoiseGit para o GIT, para o Mercurial esta disponível o TortoiseHg (Figura 5.4).

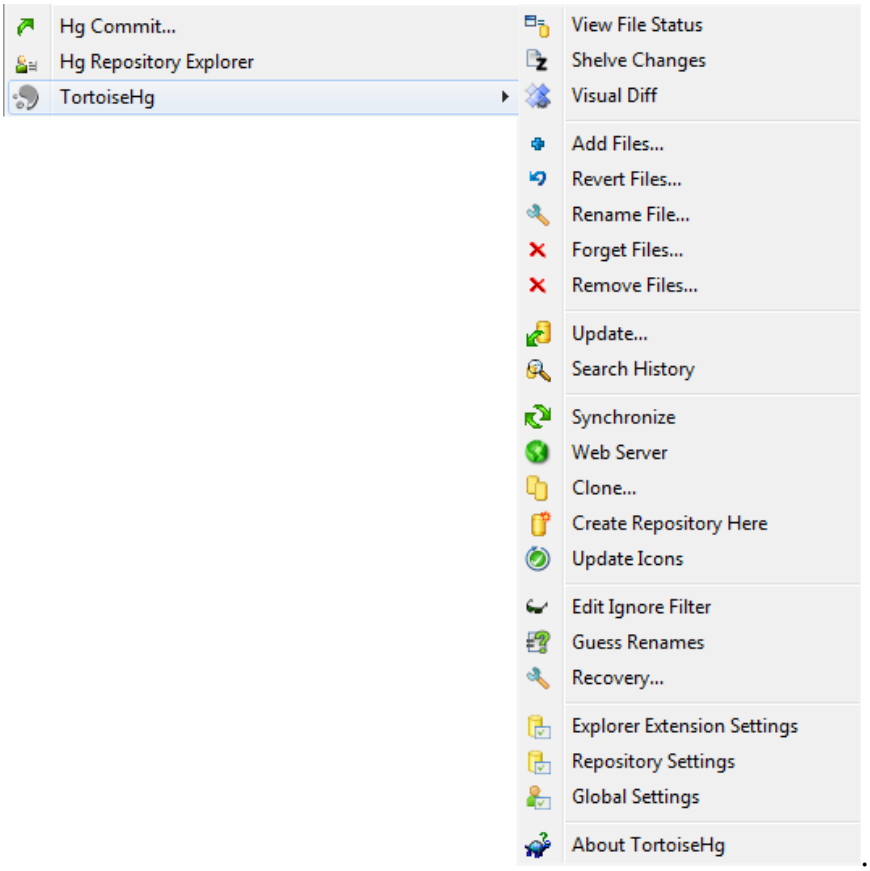


FIGURA 5.4 – Tortoisehg

5.4 CONCLUSÃO

O projeto de desenvolvimento do Mercurial quase foi um fracasso, pois o seu principal objetivo, que era substituir o Bitkeeper como ferramenta de controle de versões utilizada pelos desenvolvedores do kernel do Linux, não foi alcançado. Apesar disso, os desenvolvedores do Mercurial deram continuidade ao projeto, que pode ser considerado um sucesso pela própria popularidade alcançada pelo Mercurial.

Uma das grandes vantagens do Mercurial é a facilidade de aprendizagem que a ferramenta proporciona. O Mercurial é uma ferramenta poderosa, mas simples e mais recomendada a usuários iniciantes do que o GIT.

Outra grande vantagem do Mercurial é o fato dele ser compatível tanto com o Subversion quanto com o GIT. Isso faz com que o desenvolvedor que utiliza o Mercurial consiga realizar operações como sincronização de repositórios, clone, check out, trabalhar com ramos e outras, tanto no GIT quanto no Subversion.

Assim como os desenvolvedores do GIT, os desenvolvedores do Mercurial já conheciam bem os erros e algumas funcionalidades interessantes que não existiam tanto em SCVs centralizados como em SCVs descentralizados. Assim, eles puderam desenvolver um SCV mais completo do que seus antecessores.

6 CONSIDERAÇÕES FINAIS

A GCS surgiu devido à necessidade de uma disciplina que apoiasse o desenvolvimento de sistemas cada vez mais complexos. O sistema de controle de versões é um dos três sistemas que constituem a GCS, sob a perspectiva de desenvolvimento. Existem muitas ferramentas de controle de versões disponíveis, tanto de licença proprietária quanto de licença gratuita. Essas ferramentas atingiram um bom nível de maturidade e já vêm sendo usadas com sucesso em muitos projetos importantes.

Com um grande número de ferramentas de controle de versões, cada uma com suas características e funcionalidades, é difícil para um gerente de projetos definir qual delas é a melhor a ser adotada.

Este trabalho apresentou três importantes ferramentas de controle de versões, apresentando o histórico e uma visão geral sobre elas, com base na formulação de características de comparação, presentes no Capítulo 1. A tabela 6.1 mostra o resultado da comparação entre as ferramentas.

TABELA 6.1 – Comparativo entre Subversion, Git e Mercurial

	SUBVERSION	**GIT**	**MERCURIAL**
Licença	Apache/BSD	GNU GPL v2	GNU GPL v2
Portabilidade	Excelente	Boa	Excelente
Interoperabilidade	Excelente	Boa	Boa
Documentação	Excelente	Boa	Boa
Implantação	Regular	Boa	Excelente
Integração de Rede	Excelente	Excelente	Excelente
Replicação Remota de Repositório	Indiretamente	Sim	Sim
Propagação de alterações de repositórios	Indiretamente	Sim	Sim
***Commits* atômicos**	Sim	Sim	Sim
Permissões de acesso ao repositório	Repositório, Diretórios, Arquivos	Repositório, Diretório, Arquivos	Repositório, Diretório, Arquivos
Histórico com informações linha a linha	Sim	Sim	Sim
Suporte a *Changeset*	Parcial	Sim	Sim
Renomear/Movimentar arquivos ou diretórios	Sim	Sim	Sim
Cópias de arquivos ou	Sim	Não	Sim

diretórios			
***Merge* inteligente após Renomear/Movimentar arquivos**	Não	Não	Sim
Monitoramento de alterações não enviadas ao repositório	Sim	Sim	Sim
Mensagens de *log* para cada arquivo do *commit*	Não	Não	Não
Trabalhar com apenas um diretório do repositório (*Check out* parcial)	Sim	Não	Não
Interface Web	Sim	Sim	Sim
Interface Gráfica de Usuário	Sim	Sim	Sim

Com base neste estudo, pode-se afirmar que pelo fato de terem surgido depois dos SCVs centralizados, os SCVs distribuídos surgiram já com algumas vantagens, pois os desenvolvedores puderam corrigir erros já conhecidos e desenvolver novas funcionalidades que os próprios usuários já relatavam como essenciais. Isso não significa que eles sejam a melhor opção em qualquer situação.

O Subversion possui uma boa documentação e muitas extensões disponíveis, que aumentam ou melhoram suas funcionalidades. Ele é a melhor ferramenta, entre as três comparadas, para equipes com desenvolvedores sem experiência com controle de versões, pois é o de menor complexidade no aprendizado, para equipes que não possuam uma distribuição geográfica complexa, pois por ser centralizado, depende de uma boa disponibilidade de banda e para equipes que não possuam um número muito elevado de desenvolvedores, pois neste caso o servidor necessita de muita capacidade de processamento.

O Git é a ferramenta com pior curva de aprendizagem, comparada às outras duas. Por isso, ele é mais recomendado a usuários avançados. O Git também foi desenvolvido visando uma ferramenta de alto desempenho para o Linux, logo, é a melhor opção caso o projeto utilize esse sistema operacional. O fato de o Git possuir compatibilidade com o Subversion, também o torna a melhor escolha para projetos que vão migrar do Subversion para um SCV distribuído, principalmente se o projeto adotar o Linux como sistema operacional.

O Mercurial possui uma melhor curva de aprendizagem do que o Git, por isso, é o mais recomendado, dentre as três ferramentas comparadas, para usuários iniciantes que necessitam de um SCV descentralizado. Projetos que utilizam o Windows e que também necessitam de um SCV descentralizado, têm como melhor opção o Mercurial, pois é a ferramenta que possui a implantação mais simples neste sistema operacional. O fato de o Mercurial possibilitar a importação de um histórico do Subversion e a exportação do histórico em formato compatível com o Subversion, além de também importar históricos do Git e do CVS, o torna a melhor ferramenta no quesito de compatibilidade com outros SCVs.

Esse trabalho contribuiu com:

- uma revisão dos conceitos básicos de GCS,
- uma revisão sobre os principais conceitos de sistemas de controle de versões,
- um estudo sobre as principais funcionalidades e características de três importantes ferramentas de controle de versões,
- uma comparação entre três ferramentas de controle de versões, visando um apoio no processo de escolha entre uma delas.

Por fim, são sugeridos como trabalhos futuros a implementação, no Subversion, Git e Mercurial, das funcionalidades que estão listadas na tabela de comparação e que essas ferramentas não oferecem suporte.

Também é sugerido como trabalho futuro, estabelecer uma forma mais adequada para a GC de documentos semi-estruturados, já que os SCVs geralmente tratam linhas como unidades de comparação para qualquer arquivo texto.

Referências

BOLINGER, D., BRONSON, T. Applying RCS and SCCS: From Source Control to Project Control, 526p, Pub. O'Reilly Media, September, 1995.

CHACON, S. Pro Git. 288p. Pub. Apress, August, 2009.

CHRISTENSEN, M. J., THAYER, R. H. The Project Manager's Guide to Software Engineering's Best Practices. In: IEEE COMPUTER SOCIETY PRESS AND JOHN WILEY & SONS. 2002.

CONRADI, R., WESTFECHTEL, B. Version Models for Software Configuration Management. In: ACM COMPUTING SURVEYS, vol. 30, June, 1998.

DIAS, A. F. Conceitos Básicos de Controle de Versão de Software - Centralizado e Distribuído. 2009. Disponível em http://www.pronus.eng.br/artigos_tutoriais/gerencia_configuracao/conceitos_basicos_controle_versao_centralizado_e_distribuido.php?pagNum=0. Último acesso em outubro de 2010.

ESTUBLIER, J., LEBLANG, D., CLEMM, G., CONRAD, R., TICHY, W., HOEK, A.W. D. Impact of the research community on the field of software configuration management: summary of an impact project report. ACM SIGSOFT Software Engineering Notes, v.27 n.5, Setembro, 2002.

ESTUBLIER, J., LEBLANG, D., VAN DER HOEK, A., et al., "Impact of Software Engineering Research on the Practice of Software Configuration Management", ACM Transactions on Software Engineering and Methodology (TOSEM), v. 14, n. 4 (Outubro), pp. 1-48, 2005.

HASS, A. M. J. Configuration Management Principles and Practices Boston, MA, Pearson Education, Inc., 2003.

IEEE, Std 610.12 - IEEE Standard Glossary of Software Engineering Terminology, Institute of Electrical and Electronics Engineers, 1990.

IEEE, Std 828 - IEEE Standard for Software Configuration Management Plans, Institute of Electrical and Electronics Engineers, 2005.

LEON, A. A Guide to Software Configuration Management Norwood, MA, Artech House Publishers. 2000.

MASON, M. Pragmatic Version Control: Using Subversion (The Pragmatic Starter Kit Series). Pragmatic Bookshelf, 2006.

MURTA, L. G. P. Gerência de configuração no desenvolvimento baseado em componentes. 213p. Tese de Doutorado, COPPE, UFRJ, Rio de Janeiro, Brasil, 2006.

O'SULLIVAN, B. Mercurial: The Definitive Guide, 288p, Pub. O'Reilly Media, June, 2009.

SOMMERVILLE, I. Engenharia de Software, São Paulo, Ed. Pearson Education, 2003.

SPAIN, P. Document Version Control with CVS. 2001. Disponível em http://www.xpro.com.au/Presentations/UsingCVS/Document%20Version%20Control%20with%20CVS.htm. Último acesso em dezembro de 2010.

STALLMAN, R., Apresentação realizada na segunda conferência internacional GPLv3, Porto Alegre, Brasil. 2006. Disponível em http://fsfe.org/projects/gplv3/fisl-rms-transcript.en.html#liberty-or-death. Último acesso em novembro de 2010.

SUSSMAN, B. C., FITZPATRICK, B. W., PILATO, C. M. Version Control With Subversion For Subversion 1.6. 2009. Disponível em http://svnbook.red-bean.com/nightly/en/index.html. Último acesso em novembro de 2010.