Tribunal de Justiça do Estado do Amazonas
Divisão da Tecnologia da Informação e Comunicação
Setor de Desenvolvimento de Sistemas

Projeto: Recuperação de Desastres Portais Web TJAM
Tarefa: Pesquisa sobre o sistema GIT SCM DVCS
Local / Data: Manaus, 12 de Janeiro de 2013

GIT SCM: Sistema de Controle de Versionamento Distribuído

Sumário

Introdução

Um sistema de controle de versão é um sistema que registra mudanças em um conjunto de arquivos com o passar do tempo, de forma que seja possível reaver suas versões anteriores.

Se você é desenvolvedor Web, por exemplo, e deseja manter versões anteriores de um componente, ou do projeto inteiro, um sistema de controle de versão (VCS) é uma boa solução. Com ele, além de permitir reverter código fonte para um estado anterior, também é possível comparar as mudanças em função do tempo, visualizar quem modificou algo que esteja causando problemas, comparar versões, etc. Usando um VCS significa que, caso alguma mudança tenha desestabilizado o sistema, ou algum código fonte foi perdido, é possível recuperá-lo com facilidade.

A palavra-chave evidente ao utilizarmos VCS é ESTADO. Falando novamente do escopo do desenvolvedor Web, não é difícil imaginar como utilizar o VCS para implementar uma solução de detecção de ESTADO do código fonte em um determinado serviço Web que esteja "de cara" para a Internet, portanto, sempre vulnerável a alguma investida de alteração ou implantação de código malicioso. Por mais cuidadoso que o desenvolvedor seja, por mais segura que seja a infraestrutura de software e hardware relativos aos serviços Web, nada é invulnerável. Então, nada mais justo que seja possível detectar quais arquivos fontes foram alterados, e a possibilidade de reversão para um ESTADO anterior ao "ataque", sem a necessidade de realizar uma restauração de backup.

Basicamente, existem três paradigmas de VCS: local, centralizado e distribuído.

Apesar da simplicidade, o paradigma local não é muito confiável, pois permite ao seu utilizador incorrer em erros desastrosos, como acidentalmente sobrepor um arquivo ou diretório, e, logicamente, não permite a colaboração de terceiros no mesmo projeto. Para resolver este problema, os VCS centralizados foram criados. Tais sistemas, como o CVS, Subversion e Perforce, possuem um repositório remoto contendo todos os arquivos versionados, com os quais os clientes verificam a última versão a partir de um ponto central. Entretanto, o grande problema do paradigma centralizado reside no fato de que, se algo der errado com o repositório; uma falha no sistema de arquivos, ou na base de dados, todo o histórico das versões estará perdido.

Tanto o paradigma local quanto o centralizado sofrem deste mal: ter tudo em um só local é um grande risco. Para resolver este problema, existe o paradigma distribuído (DVCS), como é o caso dos sistemas GIT, Mercurial, Bazaar ou Darcs. Os clientes não apenas verificam a última versão dos arquivos; eles "clonam" o repositório do projeto no qual o desenvolvedor está colaborando. Isso quer dizer que, caso o servidor-repositório não esteja mais disponível, qualquer um dos clientes colaboradores poderá restaurar o(s) histórico(s) do(s) projeto(s). Existe neste paradigma um "backup natural" de todo o VCS.

Além disso, muitos desses sistemas lidam muito bem com múltiplos repositórios, o que permite ao desenvolvedor colaborar com grupos diferentes, e de diferentes maneiras, simultaneamente dentro do escopo de um mesmo projeto. Isso permite configurar um conjunto de fluxos de trabalho impraticáveis em sistemas com paradigma centralizado, como por exemplo, um fluxo de trabalho baseado em modelo hierárquico: líder do projeto > agregador > colaborador.

GIT SCM (http://git-scm.com)

A maior diferença entre o GIT e os outros VCS é a forma como o GIT armazena o seus dados. Os outros VCS normalmente enxergam suas informações como um conjunto de arquivos e suas respectivas mudanças em função do tempo. Já o GIT lida com seus dados como um conjunto de objetos de uma espécie de **sistema de arquivos** simples. Cada vez que ocorre uma mudança no diretório de trabalho de um projeto controlado pelo GIT, ele basicamente "tira uma foto" (snapshot) desses objetos, armazenando o estado desta "foto". Caso um arquivo não tenha sido alterado, GIT não armazena o objeto novamente, apenas acrescenta um "link" ao objeto anterior idêntico ao que ele armazenou previamente. Essa característica faz toda a diferença em relação ao desempenho do GIT quando comparado a outros DVCS.

Quase todas as **operações** do GIT são **locais**. A dependência do repositório remoto com o clone local é nenhuma. Isso também contribui para seu desempenho, pois não sobrecarregará a rede com suas operações. Além do que, o desenvolvedor conta com um clone completo do repositório localmente, desacoplando-o da rede, permitindo que trabalhe sem a necessidade de estar conectado o tempo todo em sua VPN; caso esteja em trânsito, pode executar suas tarefas, e assim que estabelecer sua conectividade, "descarregá-las" para o repositório.

Cada registro do GIT leva um identificador baseado em *SHA-1 hash*, o que torna impossível mudar o conteúdo de qualquer arquivo ou diretório sem que o GIT perceba. É uma excelente característica para os desenvolvedores que desejam rigor na integridade dos códigos fontes, principalmente os códigos fontes que, de alguma forma, possam ser furtiva e maliciosamente modificados em servidores públicos, como os Webservers de Internet.

Instalando o GIT na Estação do Desenvolvedor

Linux

Uma boa distribuição linux inclui em sua biblioteca de aplicações o GIT SCM. Claro que é possível baixar o código fonte da versão mais recente diretamente do portal GIT (http://git-scm.com), compilar e instalar. A versão que acompanha a distribuição do CentOS 6 e o Ubuntu 12 é a GIT 1.7, que não deixa nada a desejar em relação a última versão: GIT 1.8.

Para instalar no CentOS 6:

```
$ yum install git-core
```

Para instalar no Ubuntu

```
$ apt-get install git-core
```

Windows

Existem várias opções para o Windows, recomenda-se a GitExtensions, disponível em http://code.google.com/p/gitextensions/. Acompanha o interpretador bash e o gitk para visualizar o histórico e os ramos do repositório local.

Para utilizar o serviço de hospedagem do GitHub como repositório público / privado, na página http://git-scm.com/downloads/guis existem opções interessantes (veja também: Github.com na pg. 9).

Configurações básicas do GIT

Existem três escopos de configurações do GIT: sistema, usuário e projeto.

- **Sistema**: Configuração que afeta todos os usuários e repositórios da máquina. A manipulação de suas variáveis é feita pela seguinte sintaxe:
  `$ git config --system nome_variável valor`
  Em sistemas Linux o arquivo fica em /etc/gitconfig, no Windows é normalmente abaixo do diretório de instalação do git: <caminho instalação>\etc\gitconfig;
- **Usuário**: Configuração que afeta apenas o usuário da máquina. A manipulação de suas variáveis é feita pela seguinte sintaxe:
  `$ git config --global nome_variável valor`
  Em sistemas Linux o arquivo fica em ~/.gitconfig, no Windows é normalmente abaixo do diretório C:\Users\<usuário>\.gitconfig ou C:\Documents and Settings\<usuário>\.gitconfig;
- **Projeto**: Configuração que afeta apenas o projeto selecionado. A manipulação de suas variáveis deve ser feita dentro do diretório de cada projeto, utilizando a sintaxe:
  `$ git config nome_variável valor`

Antes de tudo, o desenvolvedor deve se identificar, para que suas modificações, mesmo as locais, sejam registradas em seu nome, principalmente se for utilizar um repositório remoto:

```
$ git config --global user.name "nome e sobrenome"

$ git config --global user.email "endereço email"
```

As outras configurações normalmente já estão definidas após a instalação dos pacotes do GIT. Para verificar as configurações, basta utilizar a opção --list no comando config:

```
$ git config --list --global
```

É importante salientar que cada escopo de configuração mais específico sobrepõe um escopo mais genérico, então, valores definidos para um projeto sobrepõem os do usuário e os do sistema.

Estados do GIT (git status) e como alterá-los

Basicamente o GIT possui nove estados e quatro pseudo estados:

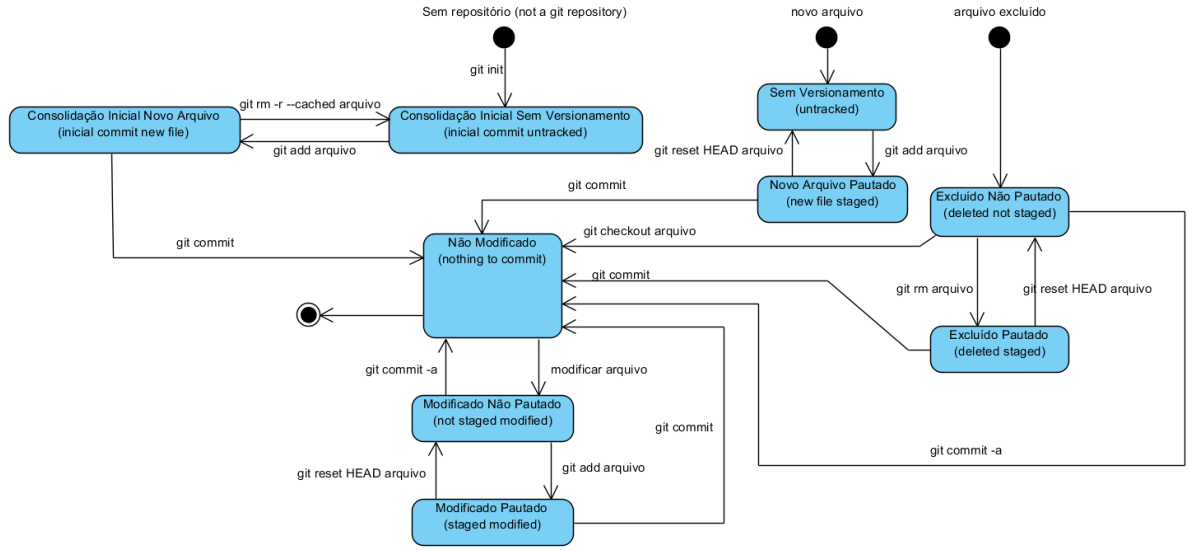

Figura 1: Conjunto dos possíveis estados dos objetos de um versionamento GIT.

Todo repositório GIT inicia no diretório no qual está o projeto a ser versionado. Entramos no diretório do projeto e:

```
$git init
```

Isso cria um diretório .git, contendo o "esqueleto" do seu repositório, ainda vazio, pois nada foi incluído no versionamento. Se for executado o comando:

```
$ git status
```

...observa-se somente o estado **Consolidação Inicial Sem Versionamento** (initial commit untracked), o primeiro estado possível dos objetos de um projeto versionado com o GIT (veja a Figura 1).

Para incluir todos os arquivos e diretórios no versionamento:

```
$ git add .
```
(diretório atual), ou, somente um diretório ou arquivo específico:

```
$ git add arquivo
```

Ao observar o git status novamente, nota-se que mudou para **Consolidação Inicial Novo Arquivo** (initial commit new file).

E efetiva-se o controle de versão nos arquivos selecionados utilizando o comando:

```
$ git commit
```

Neste momento, o GIT pede uma descrição para o comando de consolidação abrindo o editor configurado. Na realidade, a cada consolidação o GIT pede a descrição. Essa descrição é importante para que o desenvolvedor identifique o que mudou desde última consolidação.

Neste caso, trata-se de uma evidente "Consolidação Inicial". Ao salvar a descrição e sair do editor o comando é finalizado. Ao verificar o git status, o estado mudou para **Não Modificado** (nothing to commit).

A partir desse momento, os arquivos dentro do diretório de trabalho do projeto podem ser modificados, novos arquivos podem ser criados, excluídos ou movidos para outro diretório.

Ao modificar um arquivo, e verificar o git status, tem-se a mudança para o estado **Modificado Não Pautado** (not staged modified). Há dois caminhos a seguir: `$ git add arquivo`, que muda o estado deste arquivo para **Modificado Pautado**, ou `$ git commit -a`, que "pula o estado de pautado", consolidando-o imediatamente.

Ao criar um novo arquivo, e verificar o git status, tem-se um arquivo **Sem Versionamento** (untracked). Então, executa-se o `$ git add arquivo`, o qual passa para o estado **Novo Arquivo Pautado** (new file staged). E em seguinda `$ git commit` para consolidar o repositório.

Ao mover um arquivo para outro diretório, e verificar o git status, tem-se as duas situações anteriores, pois, o GIT trata a movimentação como se o arquivo fosse apagado de um diretório e criado em outro.

Ignorar arquivos e diretórios

Com frequência, depara-se com classes de arquivos onde não faz sentido ou não é necessário o versionamento, nem mesmo se deseja que o GIT mostre o estado Sem Versionamento (untracked) para tais arquivos. Arquivos de log, temporários ou gerados em tempo de execução ou compilação pelo sistema são exemplos dessa situação. Em tais casos, é possível criar um arquivo `.gitignore` contendo padrões comparativos tipo GLOB PATTERN nos quais se encaixam os arquivos indesejados.

Cada padrão deve ser colocado em uma linha separada, linhas iniciadas com # são comentários, e ignoradas. Exemplos:

```
# comentario - ingnorado
*~            # ignorar arquivos com ~ no final
!*last~       # não ignorar arquivos com last~ no final
/fazer        # ignorar o diretório raiz /fazer, não os seus subdiretórios
const/        # ignorar todos os arquivos do diretório const/
rel/*.txt     # ignorar todos os arquivos txt, mas não os que estão em subdiretórios
```

Comandos úteis (rotina básica)

Mais alguns comandos da rotina básica do versionamento:

- `$ gitk` Visualizador gráfico do histórico de consolidações realizadas no projeto.
- `$ git commit --amend` Desfaz o último commit.
- `$ git tag -a nome_etiqueta -m "descrição etiqueta"` Cria uma etiqueta de anotação.

Ramificando Versões (Branching)

Ramos (branches) são "desvios" em uma versão de um projeto, sem, contudo, afetar o caminho principal. Especificamente o GIT suporta ramificação múltipla, na qual é possível criar ramos em ramos em infinitos desvios.

Nas ferramentas de versionamento mais comuns, ramificar uma versão normalmente é um processo oneroso, exigindo uma cópia completa do projeto, o que, dependendo do tamanho do projeto, pode significar gastar tempo e espaço em disco. Essa indesejável característica não acontece com o GIT. Na realidade, o GIT utiliza um método de ramificação rápido e leve, bem diferente das outras ferramentas de versionamento, encorajando o desenvolvedor a realizar com mais frequência a operação de ramificação e mesclagem (merge).

Mas, como este método de ramificação do GIT funciona? O GIT não armazena seus dados de versionamento como uma série de conjuntos de mudanças ou diferenças (deltas), ao invés disso, ele utiliza o conceito de **snapshots**. Ao realizar uma consolidação (commit) o GIT armazena um objeto que contém um ponteiro para um snapshot do conteúdo pautado (staged), do autor e de metadados, e nenhum ou mais ponteiros para a(s) consolidação(ões) que é(são) mãe(s) desta consolidação: zero mãe para a primeira consolidação, uma mãe para uma consolidação normal, e múltiplas mães para uma consolidação resultante de uma mesclagem (merge) de um ou mais ramos.

Como ramificação é um assunto extenso, não iremos entrar em detalhes muito profundos neste material. Mas, um exemplo de uso de ramificação bem comum seriam as situações seguintes:

1. Um projeto em andamento no qual o desenvolvedor gostaria de modificar o comportamento de componente, sem, contudo, afetar a linha principal do projeto (ramo master) e, consequentemente, o trabalho dos outros desenvolvedores, os quais podem estar trabalhando com este mesmo componente;
2. O desenvolvedor cria um ramo chamado com_transparencia_v1.1 a partir do ramo master:

   ```
   $ git branch com_transparencia_v1.1
   ```

   ...e depois o seleciona como o ramo atual:

   ```
   $ git checkout com_transparencia_v1.1
   ```

3. Neste momento o desenvolvedor altera radicalmente as funções da classe modelo do componente, mudando parâmetros e escopo de funções (de pública para privada), alterando também outras classes que dependem desta, e, em seguida, testa todo o acomplamento do componente, chegando à conclusão que todo o sistema está estável:

   ```
   $ git commit -am "Modificado o componente de transparência"
   ```

4. Resolve, então, mesclar (merge) as alterações no ramo master:

   ```
   $ git checkout master
   ```
   ...para mudar para o ramo master

```
$ git merge com_transparencia_v1.1
```
...mescla o ramo com_transparencia_v1.1 no ramo atual (master)

Repositórios Remotos

Apesar de um repositório local poder ser acessado por vários desenvolvedores utilizando um compartilhamento de rede, tal como o NFS ou o SMB, não é a melhor opção de colaboração, pois, permite acesso ao sistema de arquivos do repositório, o que pode ser desastroso. O ideal é utilizar um repositório remoto para que seja possível compartilhar o trabalho realizado por vários colaboradores do mesmo projeto de maneira eficiente e segura. Além do que, um repositório remoto normalmente está em um servidor com disponibilidade 24/7, o que não é possível para uma estação de trabalho comum.

Um repositório remoto GIT nada mais é do que um repositório convencional, sem o espaço de trabalho do projeto. Um repositório remoto GIT pode ser acessado de forma "somente-leitura" ou "leitura e escrita", dependerá do protocolo utilizado, das configurações do servidor, ou do fluxo de trabalho definido para o projeto, e do nível de acesso do usuário desenvolvedor acessando o projeto.

Clonar um projeto existente de um repositório remoto

"Clonar" reflete bem a intenção distribuída do GIT: definir uma cópia local do repositório remoto no qual o desenvolvedor possa colaborar em um ou mais projetos em andamento. O clone do repositório possui as mesmas estruturas de controle do repositório remoto, adicionado a este um diretório de trabalho com os elementos (arquivos e pastas) do projeto. O comando de clonagem depende do protocolo utilizado pelo servidor do repositório remoto, normalmente http, ssh ou git, sendo este último somente-leitura; ou seja, não é possível enviar a sua contribuição com o protocolo git.

Para clonar a partir de um servidor de repositório remoto usando o **ssh**, no diretório-raiz dos projetos locais:

```
$ git clone ssh://usuario@endereço_servidor:nomeprojeto.git
```
...o "ssh://" pode ser omitido.

Para clonar a partir de um servidor de repositório remoto usando o **http**, no diretório-raiz dos projetos locais:

```
$ git clone http://endereço_servidor/nomeprojeto.git
```

Para clonar a partir de um servidor de repositório remoto usando o **git**, no diretório-raiz dos projetos locais:

```
$ git clone git://endereço_servidor/nomeprojeto.git
```

Desses protocolos, o mais seguro e eficiente é de longe o SSH. Pois, a grande maioria das distribuições Linux disponibiliza o sshd nativamente. Outra vantagem é poder definir acesso do usuário colaborador via chaves ssh, evitando fornecer senha, tornando o acesso mais seguro e ágil.

Sua autenticação passa a ser por meio do par de chaves ssh assimétricas: uma pública (no servidor do repositório), e outra privada (na estação local ~/.ssh).

Repositórios remotos privados, normalmente, definem o acesso por meio de um gestor de chaves, como o GITOSIS. Já que não usa nem usuário, nem senha, o comando de clonagem fica mais simplificado:

```
$ git clone git@endereço_servidor:nomeprojeto.git
```
...note que o usuário é sempre "git"

Para viabilizar a autenticação por chaves, o desenvolvedor anexa ao email o arquivo de sua chave pública ssh gerada pelo comando `ssh-keygen` (normalmente, um dos arquivos resultantes é id_rsa.pub), e a envia ao administrador do repositório remoto. A chave privada fica na estação de trabalho do desenvolvedor (normalmente: id_rsa). Em ambiente Windows, a aplicação "Git Bash" do GitExtensions também possui esse gerador de chaves.

Enviar um projeto local para o servidor do repositório remoto

Uma vez que o desenvolvedor tenha sua chave pública instalada no servidor do repositório, e o acesso de escrita liberado para o seu projeto, ele deve adicionar o servidor ao seu projeto local, antes de enviá-lo ao repositório remoto, sob a pasta do projeto:

```
$ git remote add origin git@endereço_servidor:nomeprojeto.git
```

Para enviar:

```
$ git origin master
```

Github.com

É um repositório na Web Internet, com suporte a SSH e HTTP, e planos de hospedagem de projetos públicos (plano sem curso) e privados (planos), e um sistema de gestão com log view, gráficos de indicadores, visualização dos ramos, Wiki, opções de Hooks, e muito mais. É uma excelente opção para equipe **interna** com opção de colaboração **externa**.

Para saber mais

Ajuda do GIT:

```
$ git help palavra-chave

$ git palavra-chave --help

$ man git-palavra-chave
```

Documentação Oficial:

http://git-scm.com/documentation

Rede social do Github:

http://github.com/

Canais #git e #github do servidor IRC irc.freenode.net