**Faculdade de Tecnologia Dr. Thomaz Novelino**

**Disciplina Eletiva – Programação Web**

**Prof. Me. Fausto G. Cintra  
Relatório Acadêmico**

**Git**

**Iury Almeida de Oliveira**

**Socrates Chieregato**

**INTRODUÇÃO**

O método preferido de controle de versão por muitas pessoas é copiar arquivos em outro diretório (talvez um diretório com data e hora, se forem espertos). Esta abordagem é muito comum por ser tão simples, mas é também muito suscetível a erros. É fácil esquecer em qual diretório o usuário está e gravar acidentalmente no arquivo errado ou sobrescrever arquivos sem querer.

O controle de versão é um sistema que registra as mudanças feitas em um arquivo ou um conjunto de arquivos ao longo do tempo de forma que o usuário possa recuperar versões específicas.

**1. O QUE É VERSIONAMENTO DE CÓDIGO?**

O controle de versão é um sistema que registra as mudanças feitas em um arquivo ou um conjunto de arquivos ao longo do tempo de forma que o usuário possa recuperar versões específicas.

Um designer gráfico ou um web designer e quer manter todas as versões de uma imagem ou layout, usar um Sistema de Controle de Versão (*Version* *Control* *System* ou VCS). Ele permite reverter arquivos para um estado anterior, reverter um projeto inteiro para um estado anterior, comparar mudanças feitas ao decorrer do tempo, ver quem foi o último a modificar algo que pode estar causando problemas, quem introduziu um bug e quando, e muito mais. Usar um VCS normalmente significa que se o usuário estragou algo ou perdeu arquivos, poderá facilmente reavê-los. Além disso, o usuário pode controlar tudo sem maiores esforços.

* 1. [SISTEMAS DE CONTROLE DE VERSÃO LOCAIS](https://git-scm.com/book/pt-br/v1/Primeiros-passos-Sobre-Controle-de-Vers%C3%A3o#Sistemas-de-Controle-de-Vers%C3%A3o-Locais)

Um método muito utilizado para controle de versão por muitas pessoas é copiar arquivos em outro diretório, porém esta abordagem é muito comum por ser tão simples, mas é também muito suscetível a erros. O problema é esquecer em qual diretório o usuário está e gravar acidentalmente no arquivo errado ou sobrescrever arquivos sem querer.

****

Figura 1 Diagrama de controle de versão local.

* 1. [SISTEMAS DE CONTROLE DE VERSÃO CENTRALIZADOS](https://git-scm.com/book/pt-br/v1/Primeiros-passos-Sobre-Controle-de-Vers%C3%A3o#Sistemas-de-Controle-de-Vers%C3%A3o-Centralizados)

Com a necessidade de trabalhar em conjunto com outros desenvolvedores, que usam outros sistemas, foram desenvolvidos Sistemas de Controle de Versão Centralizados (*Centralized* *Version* *Control* *System* ou CVCS). Esses sistemas, como por exemplo o CVS, Subversion e Perforce, possuem um único servidor central que contém todos os arquivos versionados e vários usuários que podem resgatar (checkout) os arquivos do servidor. Por muitos anos, esse foi o modelo padrão para controle de versão.

****

Figura 2 Diagrama de Controle de Versão Centralizado.

Os usuários podem ter conhecimento razoável sobre o que os outros desenvolvedores estão fazendo no projeto. Administradores têm controle específico sobre quem faz o quê. Entretanto, esse arranjo também possui grandes desvantagens. O mais óbvio é que o servidor central é um ponto único de falha. Se o servidor ficar fora do ar por uma hora, ninguém pode trabalhar em conjunto ou salvar novas versões dos arquivos durante esse período. Se o disco do servidor do banco de dados for corrompido e não existir um backup adequado, perde-se tudo. VCSs locais também sofrem desse problema, sempre que se tem o histórico em um único local, corre-se o risco de perder tudo.

* 1. [SISTEMAS DE CONTROLE DE VERSÃO DISTRIBUÍDOS](https://git-scm.com/book/pt-br/v1/Primeiros-passos-Sobre-Controle-de-Vers%C3%A3o#Sistemas-de-Controle-de-Vers%C3%A3o-Distribu%C3%ADdos)

Os Sistemas de Controle de Versão Distribuídos (*Distributed* *Version* *Control* *System* ou DVCS). Em um DVCS (tais como Git, Mercurial, Bazaar ou Darcs), os usuários não apenas fazem cópias das últimas versões dos arquivos: eles são cópias completas do repositório. Assim, se um servidor falhar, qualquer um dos repositórios dos usuários pode ser copiado de volta para o servidor para restaurá-lo. Cada checkout (resgate) é na prática um backup completo de todos os dados.

****

Figura 3 Diagrama de Controle de Versão Distribuído.

Muitos desses sistemas trabalham bem com vários repositórios remotos com os quais eles podem colaborar, permitindo que o usuário trabalhe em conjunto com diferentes grupos de pessoas, de diversas maneiras, simultaneamente no mesmo projeto. Isso permite que o usuário estabeleça diferentes tipos de workflow (fluxo de trabalho) que não são possíveis em sistemas centralizados, como por exemplo o uso de modelos hierárquicos.

**2. QUAIS AS VANTAGENS DA UTILIZAÇÃO DO VERSIONAMENTO DE CÓDIGO?**

Em termos de gestão, os DVCS contribuem para facilitar a missão das equipes de trabalho, de modo a que exerçam as suas tarefas da melhor forma, e vêm ajudar à redução de custos.

Os principais benefícios deste tipo de opção estão relacionados com a distribuição do processamento, acarretando vantagens em termos de redundância e/ou replicação de repositórios e permitindo alargar as possibilidades de colaboração e parceria entre programadores.

Benefícios para os programadores

* Rapidez: como os processos são operados localmente, deixa de ser preciso contatar o servidor central para proceder a operações como um *“Commit”*, *log* ou *diff*.
* Autonomia: permite trabalhar em modo desconectado, em qualquer local, só sendo necessária uma conexão com a rede para troca de revisões com outros repositórios.
* Ramos individuais: contrariamente aos chamados sistemas centralizados, combinar ramos não é uma obrigação para cada *“Commit”* e antes uma decisão que depende do programador; o trabalho local faz-se num ramo individual que não tem interferência com os demais ramos, mesmo em processos de sincronização entre repositórios.
* Facilidade na Fusão: os DVCS usam o rastreamento automático, o que facilita de forma significativa o processo de fusão.

Benefícios para Gestão/Coordenação

* Confiabilidade: com um sistema centralizado, qualquer problema que surja no servidor vai parar todo o trabalho de desenvolvimento. Um sistema distribuído permite que a equipa continue a trabalhar e os repositórios dos programadores servem como cópias de *backup*, não havendo riscos de perder nada do projeto.
* Redução de custos com o servidor: o repositório "central" (se ele existir) funciona como repositório "oficial", ao invés de ter uma função de processador central dos pedidos. Assim, a carga de processamento é distribuída pelas máquinas dos programadores.

**3.O QUE É O GIT?**

Git trata-se de um sistema a qual gerencia todo o processo de desenvolvimento de arquivos, ou seja, cria versões a cada modificação feita nos mesmos. Permite também que várias pessoas contribuam para um projeto simultaneamente.

4**. QUAL O FLUXO DE TRABALHO E OS PRINCIPAIS COMANDOS UTILIZADOS NO GIT DURANTE O PROCESSO DE VERSIONAMENTO DE CÓDIGO, ESPECIALMENTE NAS TAREFAS DE:**

4.1. INICIALIZAÇÃO DE REPOSITÓRIO

Para inicializar um repositório na sua máquina local, você pode começar com um novo projeto, ou um projeto já existente. Para isso você deve navegar até o diretório do seu projeto, e executar o comando:

$ git init

4.2. CLONAGEM DE UM REPOSITÓRIO JÁ EXISTENTE

Para obter os arquivos de um projeto em um servidor remoto, você deve navegar até o diretório a qual deseja obter os arquivos, e então executar o comando:

$ git clone URL do servidor remoto

4.3. *COMMIT* DE CÓDIGO DA ESTAÇÃO LOCAL PARA O REPOSITÓRIO REMOTO

Você pode propor mudanças (adicioná-las ao Index) usando os seguintes comandos “git add <arquivo>”, “git add .”.

Este é o primeiro passo no fluxo de trabalho básico do git. Para realmente confirmar estas mudanças (isto é, fazer um *“Commit”*), use “git *“Commit”* -m <comentários das alterações>”.

Agora o arquivo é enviado para o HEAD, mas ainda não para o repositório remoto.

4.4. COMO ATUALIZAR UM REPOSITÓRIO LOCAL COM O CÓDIGO MAIS RECENTE DO REPOSITÓRIO REMOTO

Para atualizar seu repositório local com a mais nova versão, basta executar o comando “git pull” na pasta de trabalho do usuário para obter e fazer o merge (mesclar) alterações remotas. Para fazer merge de um outro “*Branch*” ao seu “*Branch*” ativo (ex. master), use “git merge <“*Branch*”>”.

Em ambos os casos o git tenta fazer o merge das alterações automaticamente. Infelizmente, isto nem sempre é possível e resulta em conflitos. Você é responsável por fazer o merge estes conflitos manualmente editando os arquivos exibidos pelo git. Depois de alterar, você precisa marcá-los como merged com “git add <arquivo>”.

Antes de fazer o merge das alterações, você pode também pré-visualizá-as usando “git diff <“*Branch*” origem> <“*Branch*” destino>”

5. **O QUE É E COMO FUNCIONAM AS TAREFAS DE:**

5.1. FORK DE REPOSITORIO

Quando um desenvolvedor deseja contribuir com o projeto de outrem, então é executado o “*Fork* de repositório”. O “*Fork*” copia o repositório do dono para o repositório do contribuinte, assim possibilitando que o contribuinte realize quaisquer alterações e/ou inserções no projeto. O contribuinte pode fazer *“Commits”* e “*Push’s*” normalmente, e ao término solicitar ao dono que implemente tais alterações e/ou inserções.

5.2. CRIAÇÃO DE MANUTENÇÃO DE BRANCHES

As “*Branches*” são como uma “cópia espelho” do projeto, por exemplo, um desenvolvedor está trabalhando em um projeto sobre a “*Branch*” master (geralmente é o padrão) e outro desenvolvedor deseja contribuir para esse projeto e deseja incluir novos arquivos, novas funcionalidades, mas não deseja trata-lo como oficial ainda. Neste caso, o contribuinte efetua o “*Fork*” e a clonagem do projeto original, e então cria uma nova “*Branch*” (alternativa) para que possa efetuar quaisquer alterações e/ou inserções que desejar sem comprometer o projeto original. Ao término do trabalho, o desenvolvedor contribuinte efetua um “Merge” para incluir tudo o que ele desenvolveu na “*Branch*” alternativa à “*Branch*” master, e então solicita um “pull request” para o dono do projeto original. O dono do projeto analisa o que foi feito, e decide incluir ou não tais alterações e/ou inserções.

5.3. PULL REQUEST

Os pedidos de extração permitem que você conte aos outros sobre as alterações que você enviou para um repositório no GitHub. Uma vez que uma solicitação é aberta, você pode discutir e analisar as possíveis alterações com os colaboradores e adicionar confirmações de acompanhamento antes que as alterações sejam mescladas no repositório.

A “*Branch*” master é a “*Branch*” com o código final do projeto, estável. Criando uma nova “*Branch*”, se você submeter o pull request para o repositório original, mas se ele não for aceito, as alterações não estarão na sua “*Branch*” master.

**6 REFERENCIAS BIBLIOGRÁFICAS**

<https://git-scm.com/book/pt-br/v1/Primeiros-passos-Sobre-Controle-de-Vers%C3%A3o>

<https://github.com/MuvucaGames/Tutoriais/wiki/%5BWIP%5D-Tutorial-completo-sobre-versionamento-com-GIT>

<https://www.growunder.com/blog/dicas/65-git-como-funciona-e-quais-as-vantagens-e-desvantagens>

<https://help.github.com/articles/about-pull-requests/>

<https://tableless.com.br/contribuindo-em-projetos-open-source-com-o-github/>

<https://git-scm.com/book/pt-br/v1/Git-Essencial-Obtendo-um-Reposit%C3%B3rio-Git>

<https://git-scm.com/book/pt-br/v1/Ramifica%C3%A7%C3%A3o-Branching-no-Git-B%C3%A1sico-de-Branch-e-Merge>