Sistemas de controle de versão

Nelson Posse Lago - CCSL-IME/USP

2021

SCV: O que e por quê

- Registro do histórico
  - Recuperar versões anteriores
  - Verificar o que mudou entre as versões **e por quê**
  - Identificar o responsável por cada mudança
- Múltiplas versões
  - Versões novas e antigas são mantidas atualizadas
  - Versões de produção e versões de desenvolvimento
  - Versões experimentais com recursos novos
  - Versões “pessoais”
- Trabalho colaborativo
  - Múltiplos usuários trabalhando sobre o mesmo código
  - Integração contínua (“manual” e automática)
  - Integração com gestão de defeitos/requisitos

MERGE

Conceitos centrais de SCV

- Repositório
  - “Lugar” (opaco) onde ficam os dados e metadados
- Revisions
  - Cada nova versão de um arquivo ou conjunto de arquivos no repositório
    - Com os respectivos metadados (autoria, data etc.)
- Working tree (árvore de trabalho)
  - Cópia de uma *revision* fora do repositório que pode ser modificada
- Checkout
  - Extrair uma *revision* (ou parte, como um único arquivo) do repositório para a árvore de trabalho

Conceitos centrais de SCV

- Commit
  - Registrar uma nova versão de um arquivo ou arquivos no repositório a partir da árvore de trabalho
  - Como um *commit* cria uma *revision*, é comum usar “*commit*” como sinônimo de *revision*
- Branches (ramos)
  - Linhas de história “paralelas”, com nomes diferentes
- Trunk (tronco)
  - O ramo “principal”
    - o que isso significa depende do projeto; pode nem existir
- Tags (etiquetas)
  - Nomes dados para *revisions* específicas
- Merge (mesclar)
  - Modificações recentes feitas em paralelo ou ramos diferentes

Históricos lineares

- Podemos simplesmente guardar cópias completas dos arquivos a cada alteração
  - Juntamente com metadados, um sistema para recuperar versões específicas, um mecanismo para comparar versões...
  - **Sequência linear de versões** ao longo do tempo
- E como garantir a linearidade do histórico quando há mais de um usuário?
  - Acesso exclusivo (checkout/checkin)
    - Aquivo fica travado enquanto alguém modifica
      - Uso concorrente pode ser simulado com *branches/merges* (ramos/mesclas)
  - Usado por alguns sistemas antigos e algumas ferramentas fechadas
  - Evita conflitos, mas atrapalha o fluxo de trabalho
  - Erros fazem os arquivos ficarem indevidamente travados

Históricos lineares

- Mas não é necessário usar acesso exclusivo
  - Trechos diferentes do arquivo: é possível mesclar
    - Raramente pode causar problemas, mas somos otimistas!
  - Alterações sequenciais: a mais nova é a correta
  - Mesmo local editado por duas pessoas: solução manual
    - O usuário é responsável por garantir a linearidade nesse caso
- Mas como distinguir alterações sequenciais de alterações paralelas realizadas por duas pessoas?
  - Sabemos qual é a versão “ancestral” (no repositório) e quais são as novas versões
    - Comando “diff3”

Solução de conflitos

É possível identificar conflitos através do histórico

“A partir da versão original, foram feitas duas modificações incompatíveis”

```
public class Hello {
    public static void main (String[] args) {
        System.out.println("Hello, world");
    }
}
```

```
public class Hello {
    public static void main (String[] args) {
        System.out.println("Hello, World");
    }
}
```

```
public class Hello {
    public static void main (String[] args) {
        System.out.println("Hello, world!");
    }
}
```

Problemas com históricos lineares

- A mesclagem se baseia nas diferenças entre a versão em um ramo e uma versão ancestral
  - Ramos são “desvios temporários” dessa história linear
  - Difíceis de mesclar
    - As mesclas dependem de um ancestral comum; quando os históricos caminham em paralelo, a chance de problemas aumenta
    - Em geral, ramos são efêmeros e a mesclagem é revisada manualmente
- Funciona!
  - Mas veremos outra opção mais à frente

Sistemas centralizados

- Se vamos ter mais de um usuário, pode ser interessante usar uma arquitetura cliente/servidor
- Vantagens
  - Uso e compartilhamento simples
  - Algoritmos simples
  - Controle de acesso simples para projetos fechados
  - Todos os usuários estão sempre em dia
- Desvantagens
  - É preciso haver backups confiáveis
  - Não funcionam sem acesso à rede
  - Pode haver problemas de desempenho
  - **Controle de acesso ruim para projetos abertos**

Sistemas centralizados e software livre

- Leitura e escrita públicas
  - Qualquer um pode colaborar :D
  - Qualquer um pode vandalizar :’(
- Somente leitura para quem não é do projeto
  - Qualquer um tem acesso ao código :D
  - Quem não é do projeto tem dificuldade para modificar o código :’(
    - Pequenas modificações são simples, mas alterações maiores não podem se beneficiar do sistema de controle de versões

Sistemas centralizados e software livre

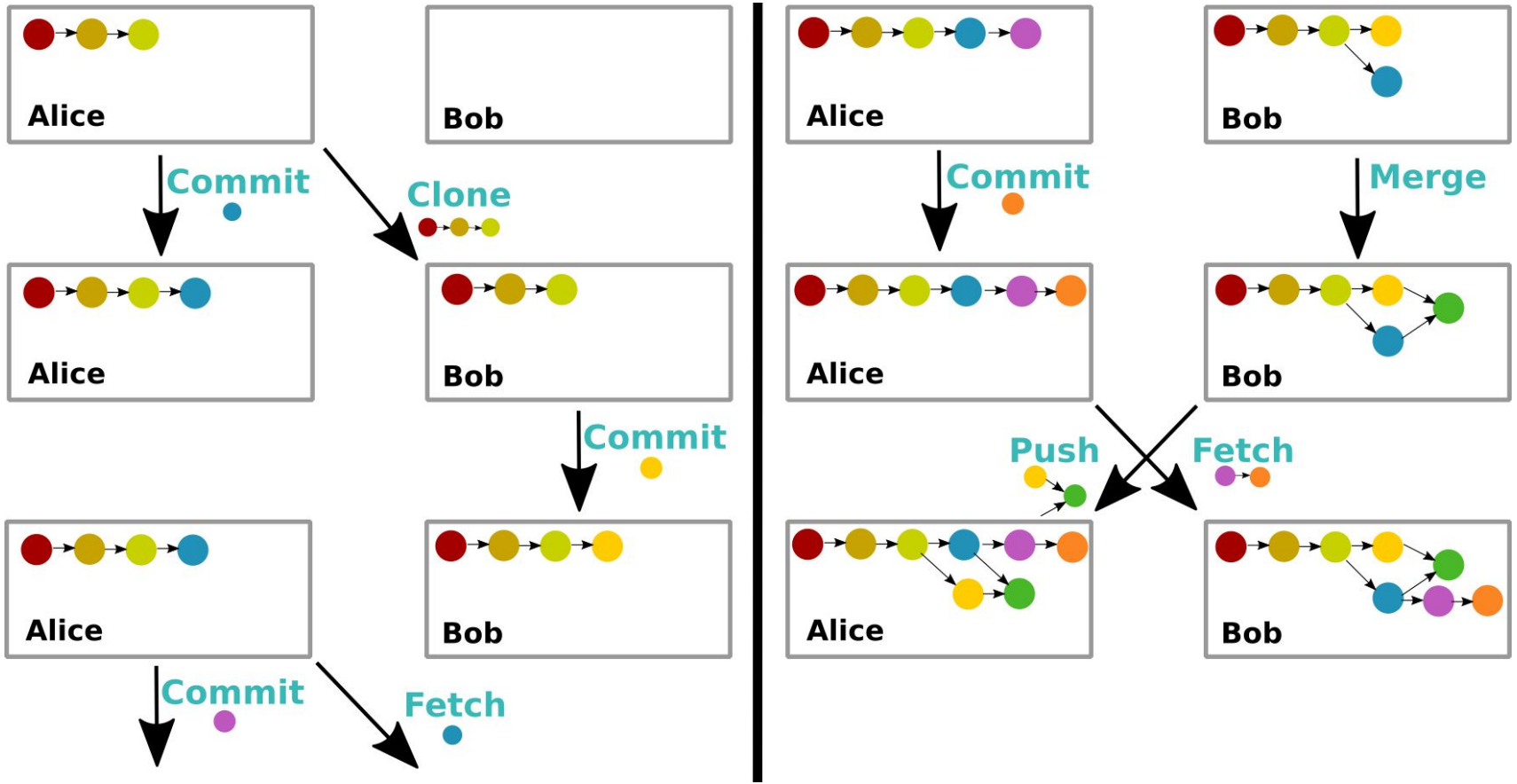
- “Solução”: patches aplicados manualmente
  - Fácil de haver erros na aplicação manual dos patches
  - Mesmo que o sistema seja capaz de usar históricos não-lineares, isso não se aplica a contribuidores externos
    - Cada patch se refere a uma versão ancestral específica
      - Se não for mesclado rapidamente, torna-se cada vez mais difícil
      - Não pode ser muito grande
  - Não há registro automático da autoria
  - Não há auxílio na gestão de conflitos
- Todos os sistemas centralizados abertos amplamente usados usam histórico linear
  - Não é possível criar versões pessoais
    - Mesmo que fosse possível guardar o histórico dessas versões, como vimos, a mesclagem seria muito difícil

Sistemas descentralizados

- Mas a história na verdade é um DAG!
  - Várias linhas paralelas, mas sempre vai “para a frente”
  - Cada *revision* pode ter mais de um ancestral
    - Mais de um ancestral indica uma mescla
  - Cada *revision* pode ter mais de um descendente
    - Mais de um descendente indica um novo ramo
- Em conflitos e mesclas, é possível considerar todo o histórico e não apenas a última versão
  - Um mesmo ramo pode ser mesclado várias vezes
  - Mesclas “grandes” são muito mais simples
  - Ramos são parte natural da história
- Mas é complicado em um sistema centralizado

Sistemas descentralizados

- E se o sistema não for cliente/servidor?
  - Todos têm uma cópia completa do repositório
    - Simplifica o processamento do DAG (por exemplo, ao mesclar)
    - O repositório tem backups “naturais”
  - Não é preciso acesso à rede para usar
  - Histórico local é possível
  - Qualquer um pode colaborar mais facilmente
    - Versões pessoais são possíveis
    - Mesclas remotas incluem metadados e podem ser semi-automáticas
  - Modelo “peer-to-peer”
    - Mas um dos *peers* pode ser um servidor



Inspirado em
https://blog.pronus.io/posts/conceitos-basicos-de-controle-de-versao-de-software-centralizado-e-distribuido/

Sistemas descentralizados

- Histórico é explicitamente não-linear
  - A ordem de commits em repositórios diferentes é diferente
  - Numerar os commits de forma sequencial não é portátil entre repositórios
    - Bazaar tenta fazer isso
  - git usa um hash sobre o conteúdo para identificar os commits
    - Cada commit tem um hash único em todos os repositórios
    - Em geral, usam-se tags e “nomes” relativos, como “HEAD~5”

Pegadinhas

- Histórico de diretórios
  - Quase todos os sistemas ignoram
    - Não é realmente necessário (o “histórico” do diretório é na verdade o histórico de seus arquivos)
    - Pode gerar situações esquisitas
  - Isso significa que nunca há diretórios vazios!
    - Às vezes queremos que um diretório vazio seja criado **na árvore de trabalho**, mas isso não tem nada a ver com versões!
      - Ainda assim, se queremos que um diretório vazio exista na árvore de trabalho, um “truque” comum é criar um arquivo vazio (“.placeholder”)
- Rename
  - Como fica o histórico?
  - E quando o rename sobreescreve um arquivo?
  - Em geral, é tratado como “apagou um arquivo e criou um novo”
  - git tenta detectar renames apenas para mostrar ao usuário

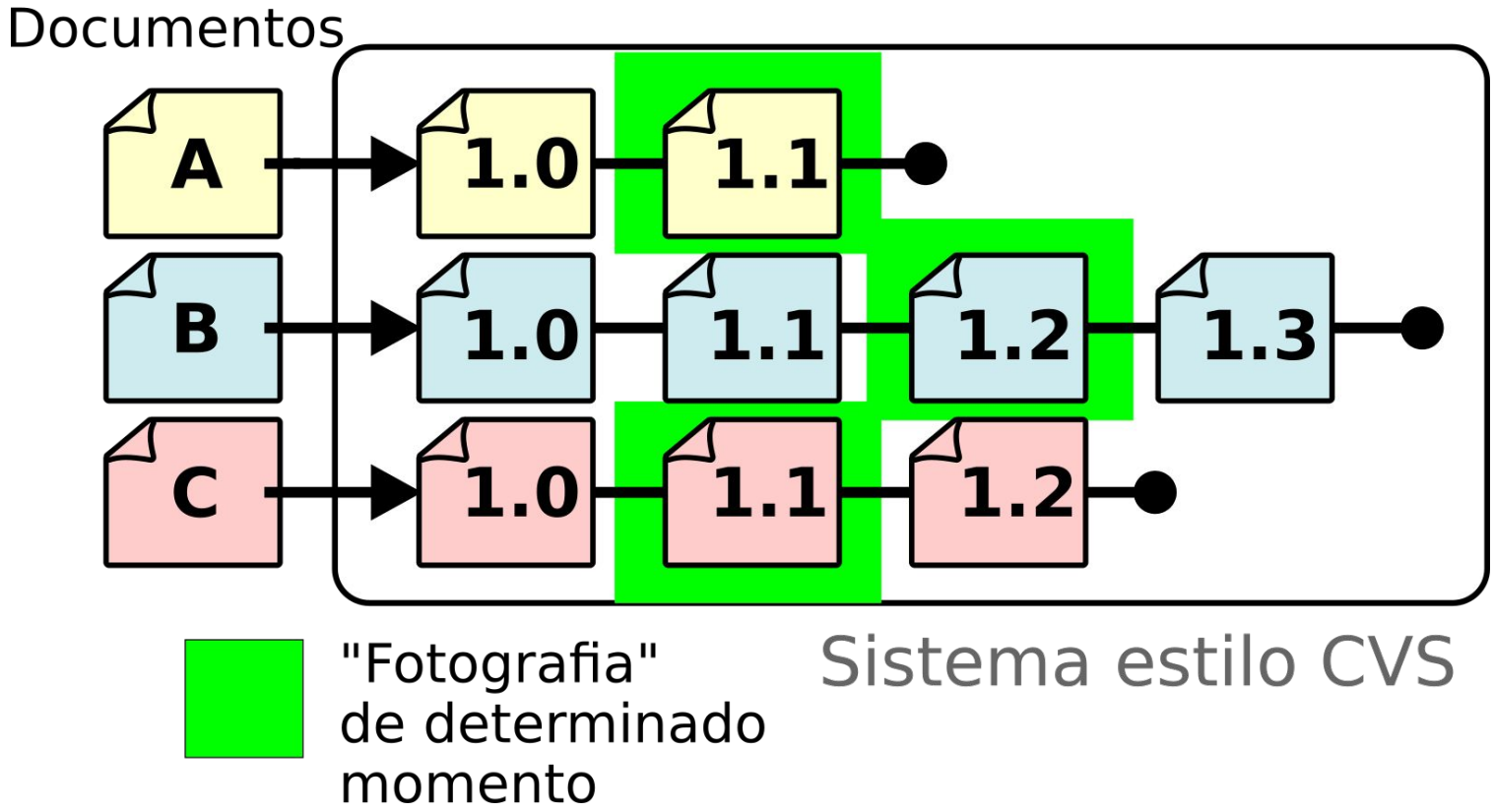
Pegadinhas

- Arquivos binários
  - Como armazenar eficientemente?
    - Em geral, cópias “rasas” quando possível e xdelta
    - Git permite guardar objetos grandes em diretório à parte
  - Como mostrar as modificações?
  - Como fazer mesclas?
- Cherry-picking
  - Tomar o patch correspondente a um commit específico e aplicá-lo em outro lugar
  - Excelente receita para dar problema!
    - Não há nenhum registro que isso foi feito; tentar fazer uma nova mescla com o ramo que contém esse commit vai resultar em conflitos
    - Num sistema descentralizado, em alguns casos é possível simular com ramos e mesclas
  - Por isso em geral é melhor ter ramos separados para cada nova funcionalidade

Sistemas abertos mais usados

- RCS
  - Apenas acesso local
  - Acesso exclusivo
  - Um único diretório
  - Ainda raramente usado para arquivos de configuração ou outras aplicações específicas por sua simplicidade
- CVS
  - Implementação trata o histórico de cada arquivo separadamente
    - Commits não são atômicos
    - Arquivos têm “números de versão” independentes
  - Diversas limitações e características indesejáveis
  - Em geral, só é usado hoje de forma legada
    - Maioria dos projetos migrou para subversion ou git

Histórico de versões (cada documento uma versão)



Sistemas mais usados (além do git)

- Subversion
  - “CVS melhorado”
    - Commits são atômicos
    - O histórico se refere a todo o repositório, com números de versão sequenciais
  - Subdiretórios podem ser baixados/sincronizados separadamente
  - Ramos “não existem”; são apenas novos diretórios com cópias do conteúdo do “ramo” do qual derivam
    - No servidor, essas cópias são “rasas”, mas na árvore de trabalho elas ocupam espaço
  - Atualmente é o sistema centralizado mais usado

Sistemas mais usados (além do git)

- Bazaar
  - Python
    - Bom suporte multiplataforma
  - Foco em usabilidade
  - Desenvolvido pela canonical
    - Está na base do launchpad
  - Durante muito tempo, desempenho problemático
  - Além da canonical, poucos projetos - e cada vez menos!
- Mercurial
  - Python
    - Bom suporte multiplataforma
  - Alguns projetos importantes usam (especialmente por conta do suporte multiplataforma)