Controle de versões

CVS

Subversion

Git

↳ Código?

☺ Dropbox, TCC_v1_b.docx

EQUIPES desenvolvem SW

SERVIDOR p/armazenar

Histórico de versões

VCS → Sistema de

Controle de Versões

VCS

repositório

recuperar versões antigas

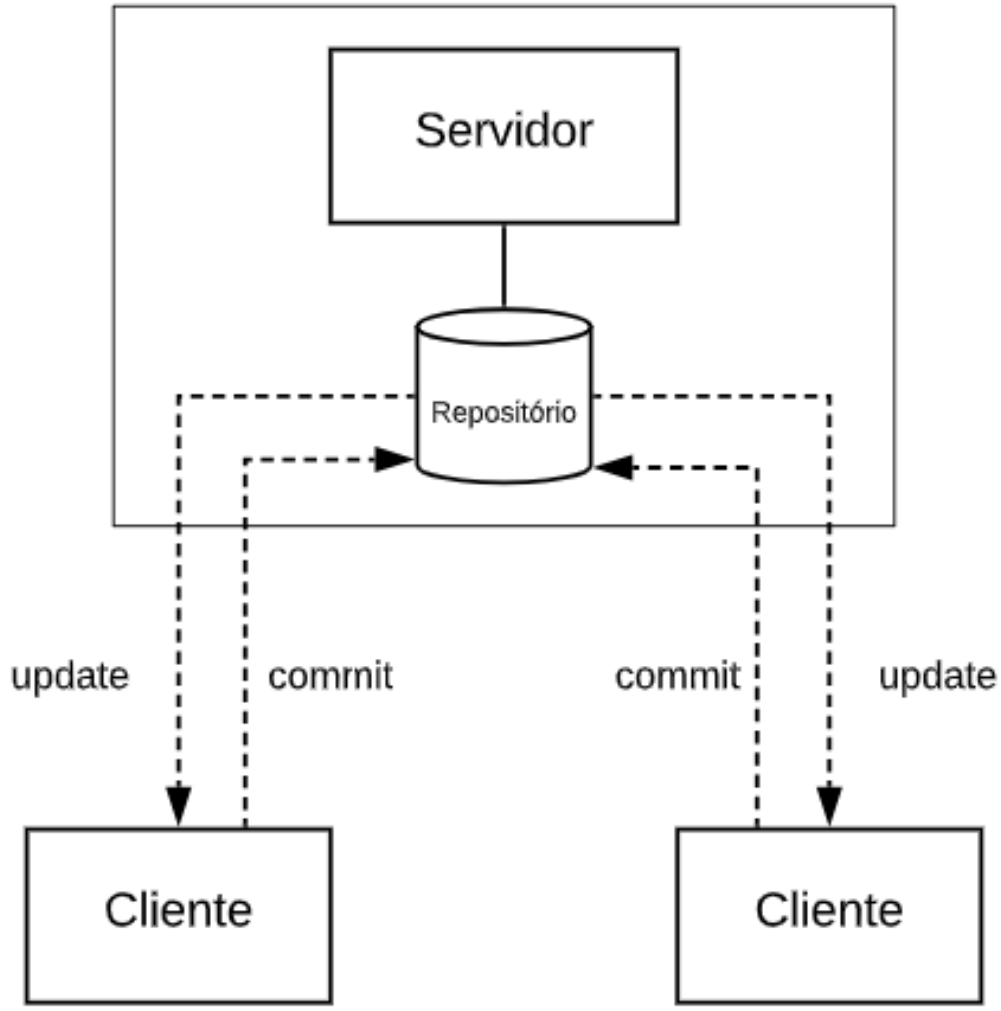
Histórico

SCCS Unix

CVS ~80s

Subversion ~2000

centralizados, cliente/servidor



VCS Centralizado. Existe um único repositório, no nodo servidor.

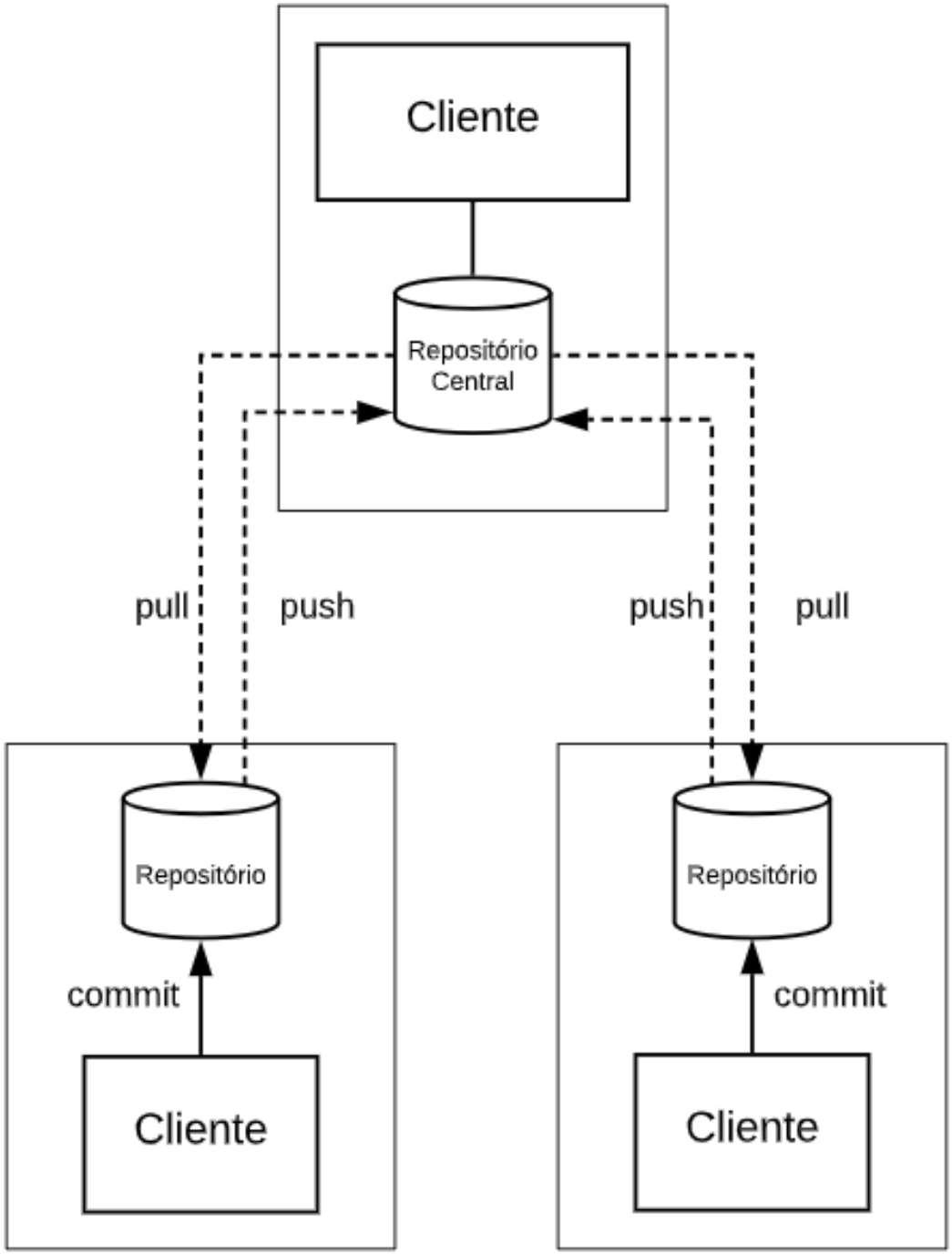
DVCS - Sists. de C. de V. DISTRIBUÍDOS

BitKeeper 2000

Mercurial e git 2005

peer-to-peer

cada desenvolvedor possui em sua máquina um servidor completo de controle de versões, que pode se comunicar com os servidores de outras máquinas



VCS Distribuído (DVCS). Cada cliente possui um servidor. Logo, a arquitetura é peer-to-peer.

Teoria → peers são equivalentes

Prática → referência
repositório central

pull — push

Vantagens DVCS

- offline
- commits cl + freq.
- commits + rápidos
- sincronização n precisa ser com o repo central

GIT

Linus Torvalds

Linux usava Bitkeeper de graça

Bk decidiu cobrar (2005)

Palestra https://bit.ly/3X3zZTt

DVCS, claro, e aberto

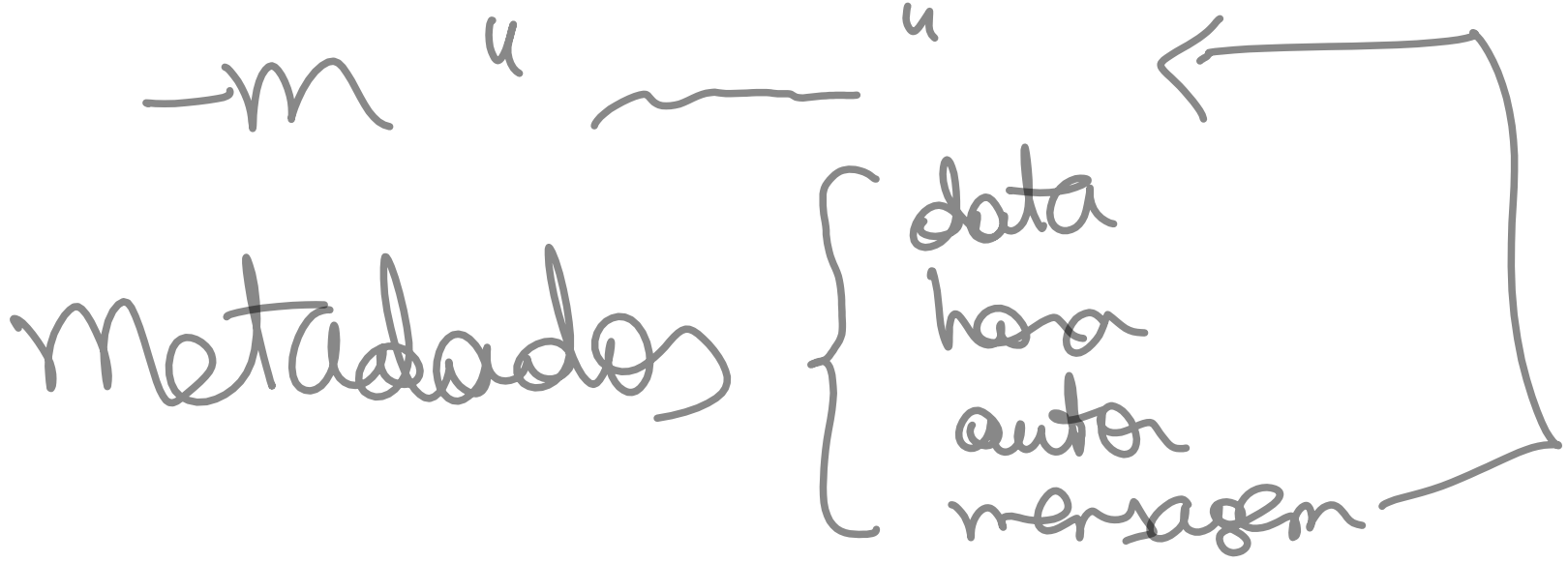
GitHub

GitLab http://gitlab.utfpr.edu.br/

Bitbucket

GIT — Controle de Versões

- init —— .gitgrore
- clone
- commit
  - m " —— "



Regras p/ commits

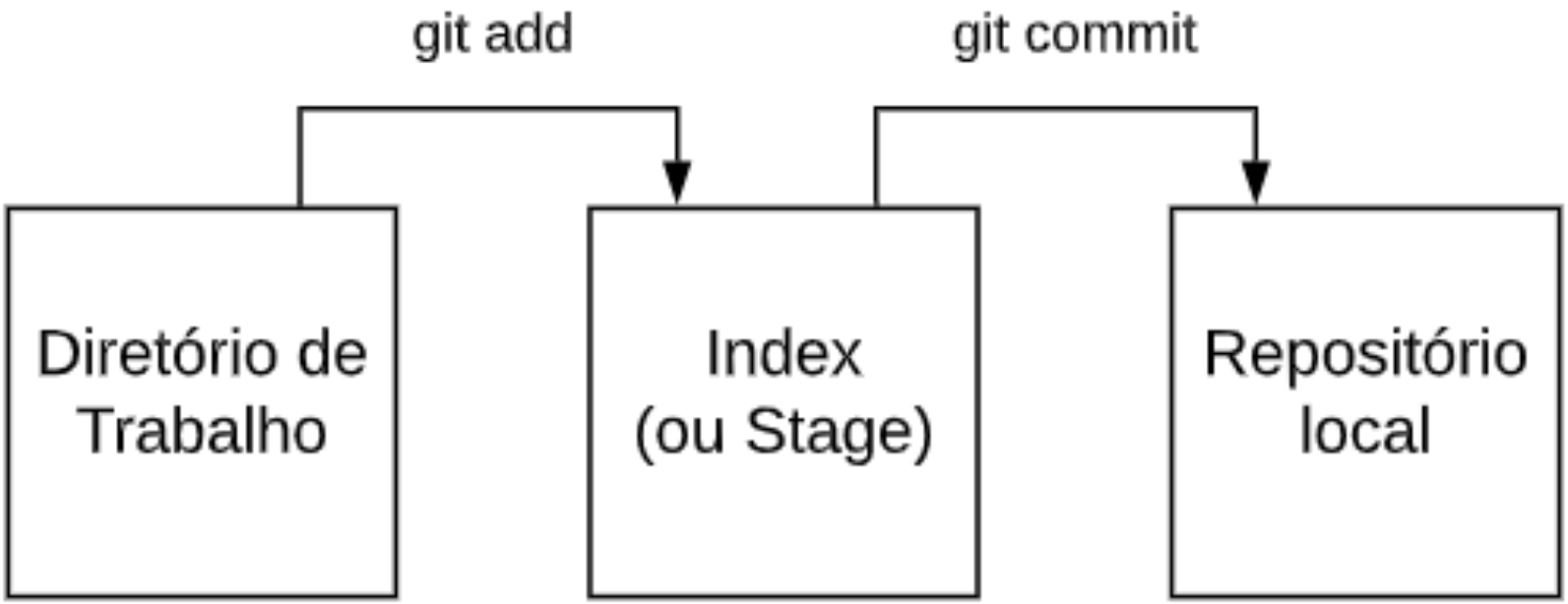
incluir apenas

mudanças relacionadas

Id único do commit

git add

Apenas os adicionados serão gerenciados



Comandos `add` e `commit`

Exemplo: Suponha o seguinte arquivo simples, mas suficiente para explicar os comandos `add` e `commit`.

```
 // arq1
x = 10;
```

Após criar esse arquivo, o desenvolvedor executou o seguinte comando:

`git add arq1`

Esse comando adiciona o arquivo `arq1` no index (ou stage). Porém, logo em seguida, o desenvolvedor modificou de novo o arquivo:

```
// arq1
x = 20; // novo valor de x
```

Feito isso, ele executou:

`git commit -m "Alterando o valor de x"`

A opção `-m` informa a mensagem que descreve o commit. Porém, o ponto que queremos ressaltar com esse exemplo é o seguinte: como o usuário não executou um novo `add` após mudar o valor de `x` para 20, a versão mais recente do arquivo não será salva pelo commit. Em vez disso, a versão de `arq1` que será versionada é aquela em que `x` tem o valor 10, pois ela é a versão que consta do index.

Para evitar o problema descrito nesse exemplo, é comum usar um `commit` da seguinte forma:

```
git commit -a -m "Alterando valor de x"
```

A opção `-a` indica que antes de executar o commit queremos adicionar no index todos os arquivos rastreados (*tracked*) que tenham sido modificados desde o último commit. Portanto, a opção `-a` não elimina a necessidade de usar `add`. O uso desse comando continua sendo necessário, pelo menos uma vez, para indicar ao git que desejamos tornar um determinado arquivo rastreável.

git rm

Da mesma forma que existe um `add`, também existe uma operação para remover um arquivo de um repositório git. Um exemplo é dado a seguir:

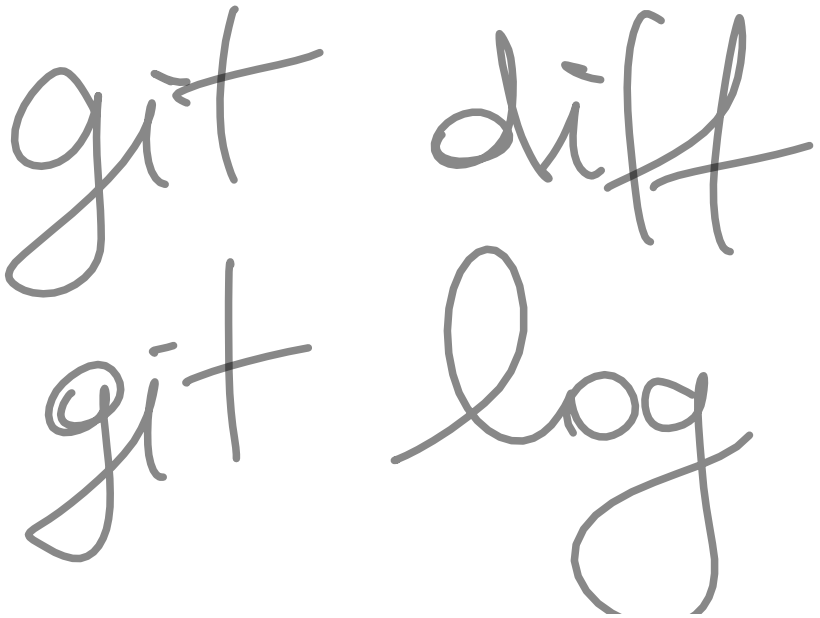
```
git rm arq1.txt
git commit -m "Removendo arq1.txt"
```

Além de remover do repositório git local, o comando `rm` também remove o arquivo do diretório de trabalho.

git status

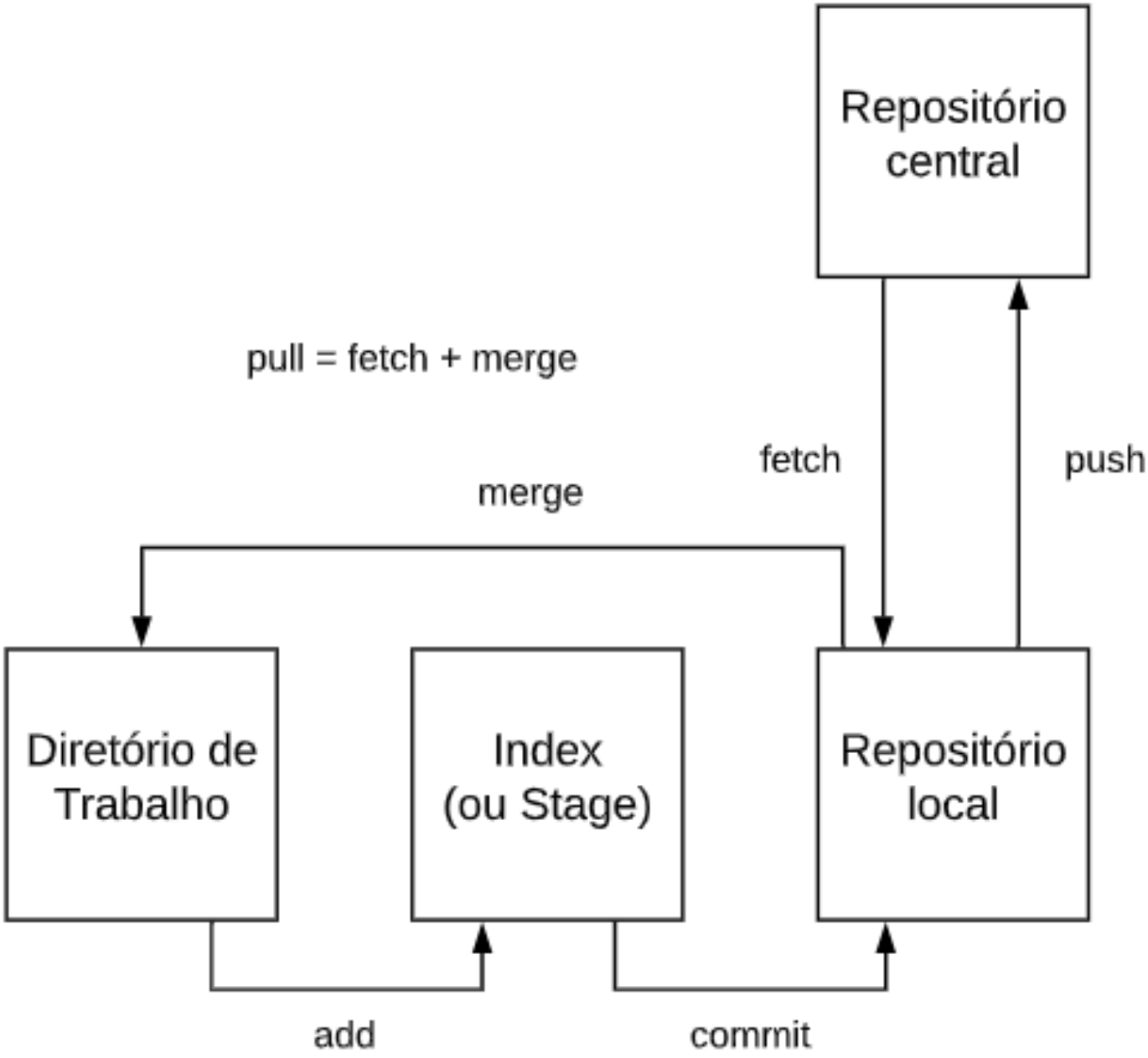
O comando **status** é um dos comandos git mais usados. Dentre outras informações, ele mostra o estado do diretório de trabalho e do index. Por exemplo, pode-se usar esse comando para obter informações sobre:

- Arquivos do diretório de trabalho que foram alterados pelo desenvolvedor, mas que ele ainda não adicionou no index.
- Arquivos do diretório de trabalho que não são rastreados pelo git, ou seja, eles ainda não foram objetos de um `add`.
- Arquivos que encontram-se no index, aguardando um `commit`.



O comando `git diff` é muito usado para destacar as modificações realizados nos arquivos do diretório de trabalho e que ainda não foram movidas para o index (ou stage). Para cada arquivo modificado, ele mostra as linhas que foram adicionadas (+) e removidas (-). Muitas vezes, usamos um `git diff` antes de um `add` / `commit` para ter certeza das mudanças que iremos "perpetuar", em seguida, no sistema de controle de versões.

Já o comando `git log` lista informações sobre os últimos commits, como data, autor, hora e descrição do commit.



Comandos `push` e `pull`

Conflitos de Merge 1.6 EM

Central	Bob	Alice
Hello, world!	Hello, world!	Hello, world!
Olá, mundo!	Hello, world!	Olá, mundo!

pull

```
main() {
<<<<<<< HEAD
print("Hello, world!");
=======
print("Olá, mundo!");
>>>>>>> f25bce8fea85a625b891c890a8eca003b723f21b
}
```

BRANCHES 1.7 ESM

master → main

git branch f-nav

git checkout [nome branch]

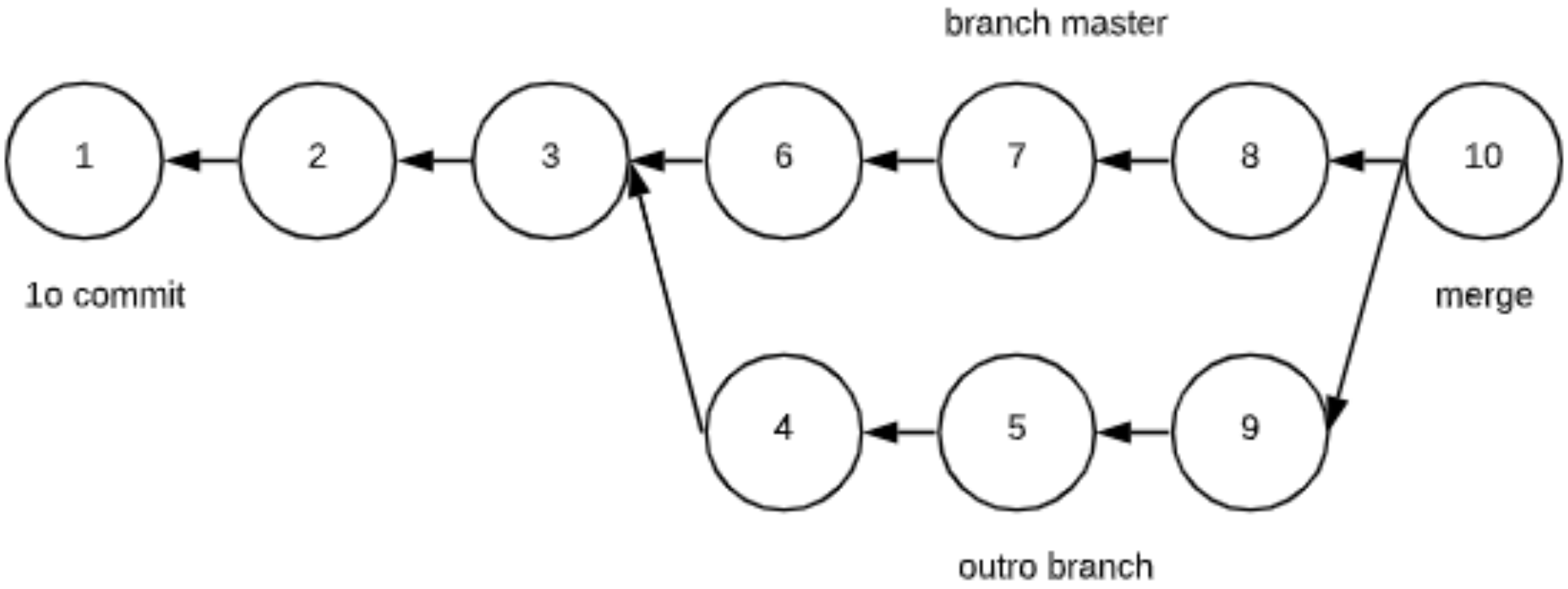
git branch

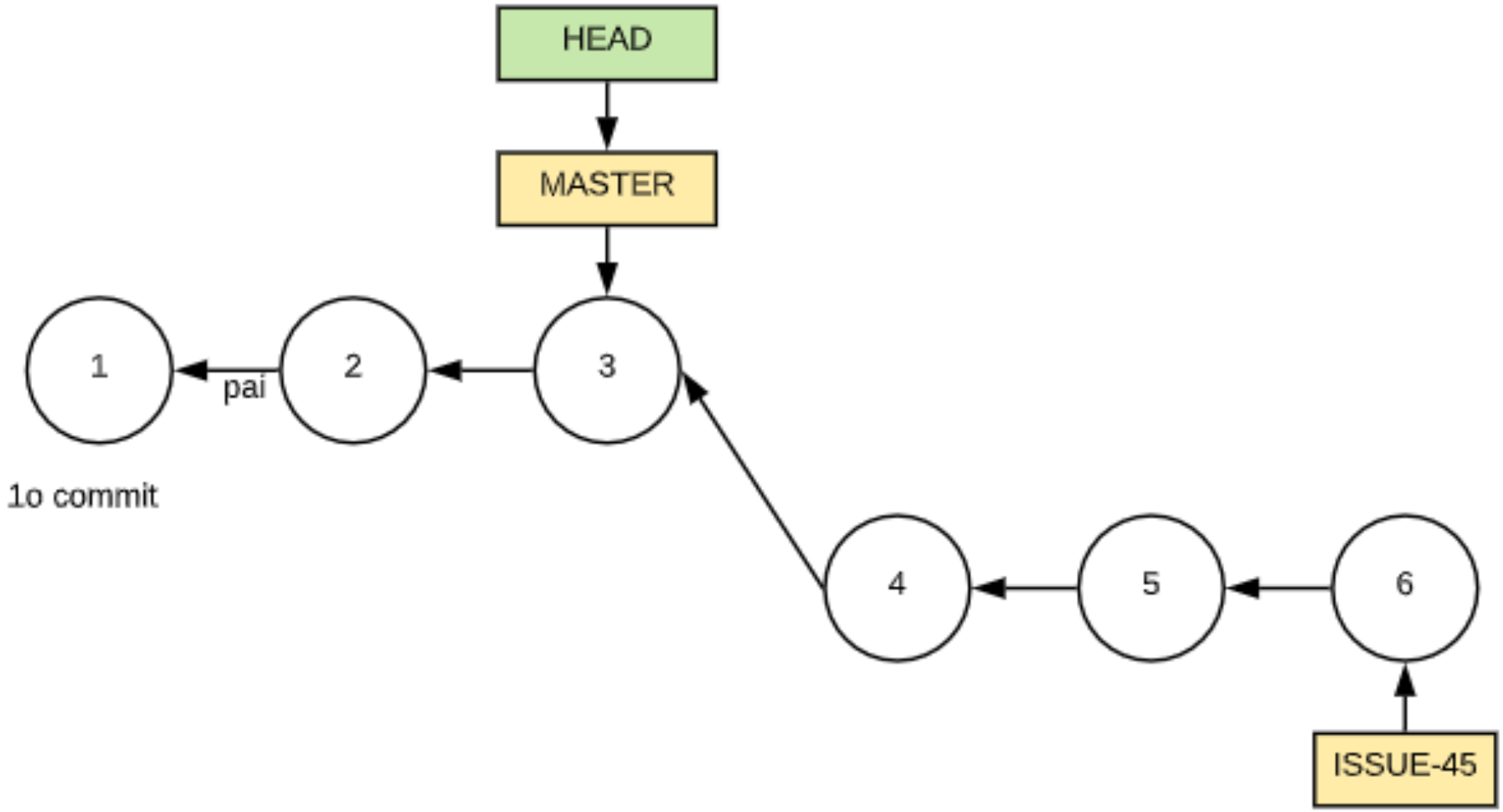
git merge f-novo

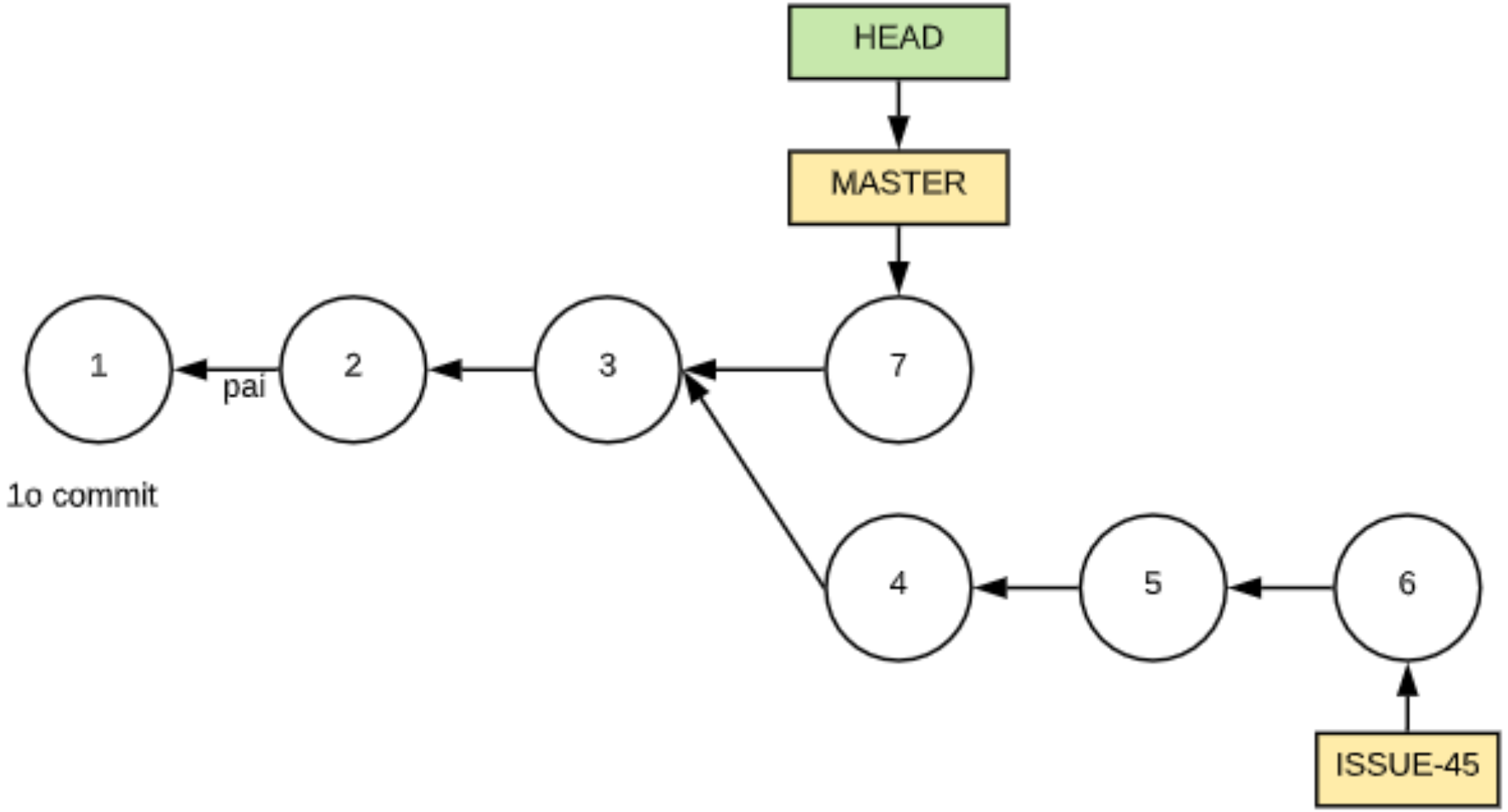
Conflitos de integração

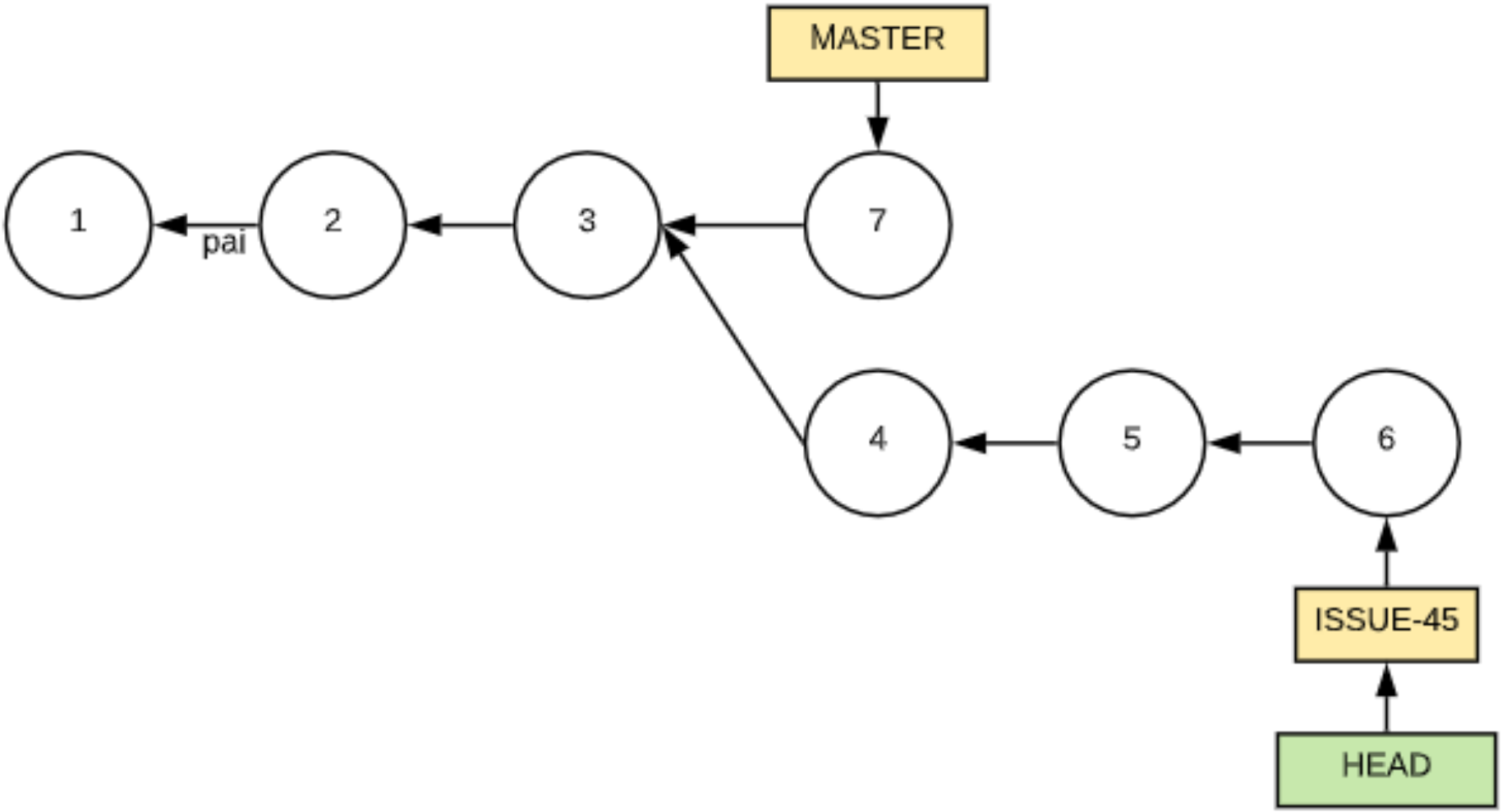
git branch -d f-novo

Grafo de Commits 1.7.1




HEAD
MASTER
1
pai
2
3
1o commit
4
5
6
ISSUE-45


HEAD
MASTER
1
pai
2
3
7
1o commit
4
5
6
ISSUE-45


MASTER
1
pai
2
3
7
4
5
6
ISSUE-45
HEAD

Branches Remoto 18

Exemplo: Suponha que Bob criou um branch, chamado `g-novo`, para implementar uma nova funcionalidade. Ele realizou alguns commits nesse branch e agora gostaria de compartilhá-lo com Alice, para que ela implemente parte da nova funcionalidade. Para isso, Bob deve usar o seguinte `push`:

```
git push -u origin g-novo
```

Esse comando realiza o `push` do branch corrente (`g-novo`) para o repositório remoto, chamado pelo git de `origin`. O repositório remoto pode, por exemplo, ser um repositório do GitHub. O parâmetro `-u` indica que, no futuro, vamos querer sincronizar os dois repositórios por meio de um `pull` (a letra do parâmetro vem da palavra *upstream*). Essa sintaxe vale apenas para o primeiro `push` de um branch remoto. Nos comandos seguintes, pode-se omitir o `-u`, isto é, usar apenas `git push origin g-novo`.

No repositório remoto, será criado um branch `g-novo`. Para trabalhar nesse branch, Alice deve primeiro criá-lo na sua máquina local, mas associado ao branch remoto, por meio dos seguintes comandos, que devem ser executados no master:

```
git pull
```

```
git checkout -t origin/g-novo
```

O primeiro comando é necessário para tornar o branch remoto visível na máquina local. Já o segundo comando cria um branch local, chamado `g-novo`, que Alice vai usar para rastrear mudanças no branch remoto `origin/g-novo`, conforme indica o parâmetro `-t`, que vem da palavra *tracking*. Em seguida, Alice pode realizar commits nesse branch. Por fim, quando estiver pronta para publicar suas mudanças, ela deve executar um `push`, com a sintaxe normal, isto é, sem o parâmetro `-u`.

Agora, Bob pode realizar um `pull`, concluir que a implementação da nova funcionalidade está finalizada e, portanto, pode ser integrada no master, por meio de um merge. Bob pode também deletar os branches local e remoto, usando os comandos:

```
git branch -d g-novo
```

```
git push origin --delete g-novo
```

E Alice também pode deletar seu branch local, chamando apenas:

```
git branch -d g-novo
```

Pull Requests

Squash

```
git rebase -i HEAD~5
```

Forks