Versionado de código

Versionado de código

Permite realizar un seguimiento de los cambios realizados en el código fuente a lo largo del tiempo.

Los primeros sistemas de versionado de código (SCV) surgieron en la década de 1970, como respuesta a la necesidad de gestionar y controlar los cambios en el código fuente de los programas informáticos.

Algunas ventajas de su uso:

- **Seguimiento de cambios:** permite registrar y auditar todos los cambios realizados en el código fuente.
- **Colaboración:** Permite a varios desarrolladores trabajar en el mismo proyecto sin sobrescribir el trabajo de los demás.
- **Resolución de errores:** Facilita la identificación y reversión de errores.
- **Mantenimiento del proyecto:** Permite mantener un registro de los cambios realizados en el código, lo que facilita la comprensión del proyecto y la realización de cambios en el futuro.

Sistemas de control de versiones

- **Centralizados:** Almacenan todos los archivos del proyecto en un único servidor. (SVN, CVS).
- **Distribuidos:** Cada desarrollador tiene una copia completa del repositorio (todo el código y toda la historia) en su propia pc. El ejemplo más conocido es Git.

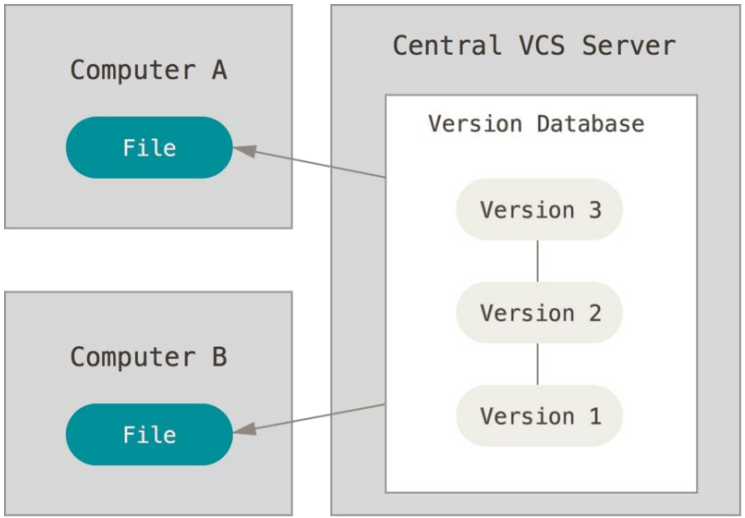
Git

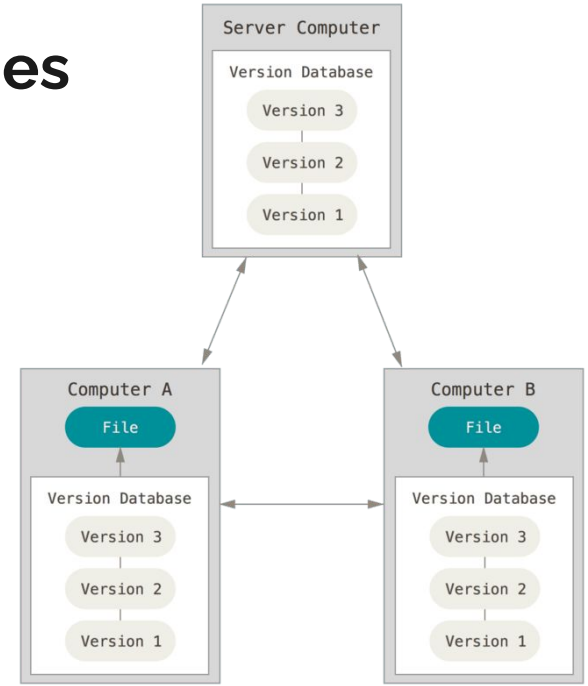
Creado por Linus Torvalds (el creador del kernel de Linux) en 2005 como una herramiento de control de versiones para el desarrollo del kernel de Linux.

Inspirado en otros sistemas de control de versiones como Mercurial y BitKeeper. Se desarrolló rápidamente con la colaboración de otros desarrolladores de la comunidad Linux.

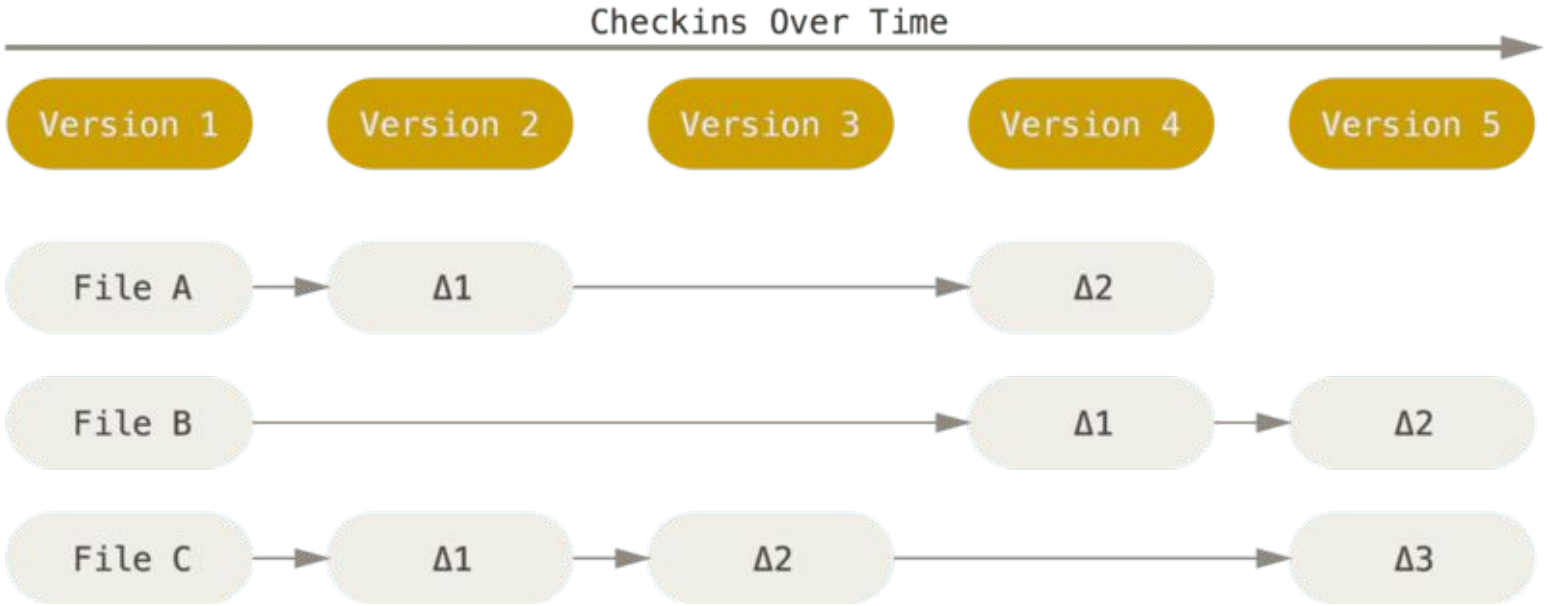
Su eficiencia, flexibilidad y enfoque de código abierto lo impulsaron a convertirse en el sistema de control de versiones dominante en el mundo del desarrollo.

Sistemas de control de versiones

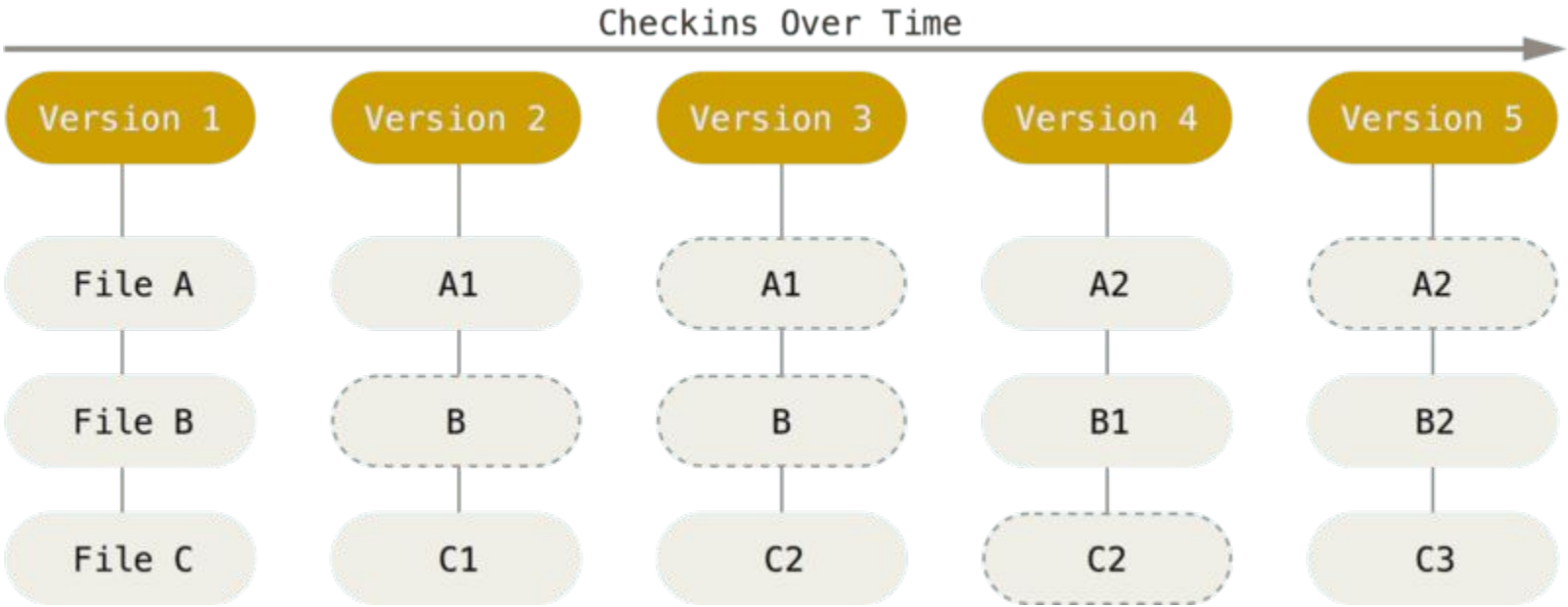




Control de versiones basados en cambios (delta-based)



Control de versiones basados en Snapshots



¿Cómo funciona git?

- Almacena los archivos en un repositorio (pares clave-valor)
- Cada cambio se guarda como un "commit"
- Los commits se pueden ramificar y fusionar (branch, merge)

¿Cómo arrancar un repositorio?

1) **Desde cero:**

   ~$ mkdir proyecto/

   ~$ cd proyecto/

   ~/proyecto$ git init

   Initialized empty Git repository in /home/erica/proyecto/.git/

¿Cómo arrancar un repositorio?

2) Clonarlo:

```
~$ git clone https://github.com/torvalds/linux
Cloning into 'linux'
remote: Enumerating objects: 8716629, done.
remote: Total 8716629 (delta 0), [...]
Receiving objects: 100% [...]
Resolving deltas: 100% [...]
Updating files: 100% [...], done.
```

Viendo el estado del repo..

~/proyecto$ git status

On branch master

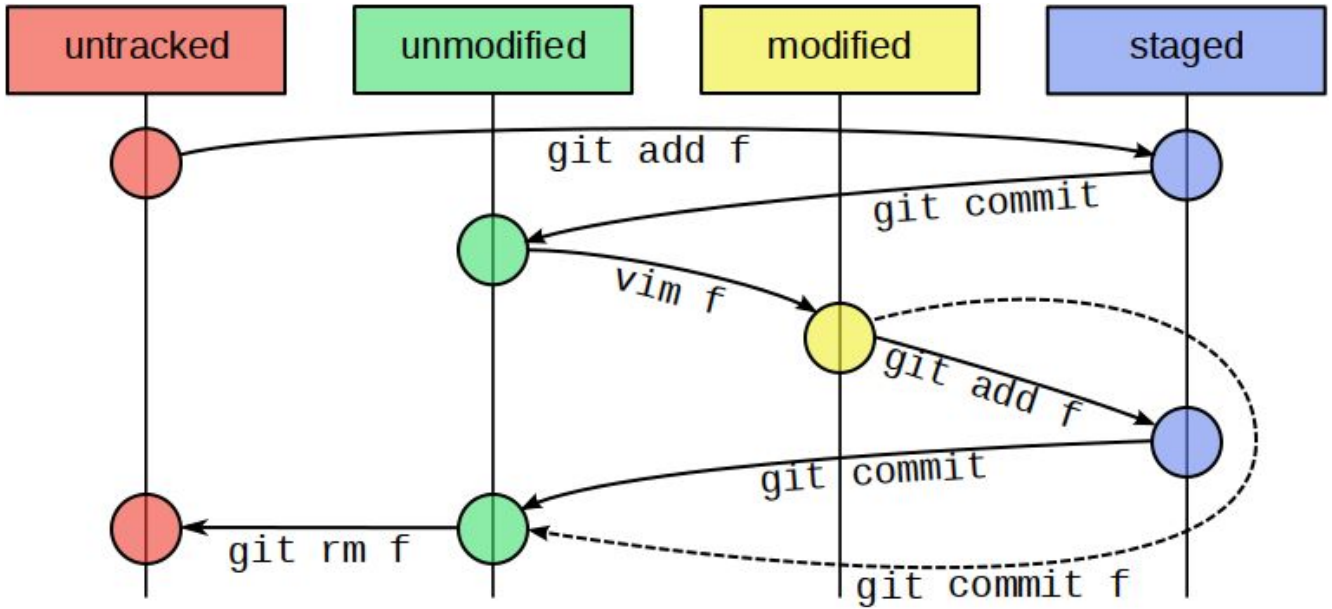
No commits yet

nothing to commit (create/copy files and use "git add" to track)

Viendo el estado del repo..

```
echo 'int main(){return 0;}' > main.c

~/proyecto$ git status

On branch master

No commits yet

Untracked files:

 (use "git add <file>..." to include in what will be committed)

   main.c

nothing added to commit but untracked files present (use "git add" to track)
```

Estados


untracked
unmodified
modified
staged
git add f
git commit
vim f
git add f
git commit
git rm f
git commit f

Staging area

~/proyecto$ git add main.c

~/proyecto$ git status

On branch master

No commits yet

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: main.c

Commit

```
~/proyecto$ git commit -m "Inicio proyecto"

[master (root-commit) 4fdfd08] Inicio proyecto
 1 file changed, 1 insertion(+)
 create mode 100644 main.c
```

Log

~/proyecto$ git log

commit 4fdfd088bb1f3df9413aaba557448f92b8f134b5 (HEAD -> master)

Author: Erica Vidal <ericavidal@gmail.com>

Date: Mon Mar 18 14:54:38 2024 -0300

    Inicio proyecto

~/proyecto$ git log –-stat

~/proyecto$ git log -p

~/proyecto$ git log show

¿Qué es un commit?

Un commit en Git es un snapshot del estado del proyecto en un momento determinado.

- **Almacena los cambios:** Un commit guarda los cambios realizados en los archivos del proyecto desde el último commit.
- **Incluye un mensaje:** Cada commit debe tener un mensaje asociado que describa brevemente los cambios realizados. Escribir mensajes de commit claros y descriptivos es una buena práctica para facilitar la comprensión del historial del proyecto en el futuro (lo vemos más adelante).
- **Crea un registro histórico:** Los **commits se encadenan cronológicamente**, formando un registro histórico de todos los cambios realizados del proyecto.
- **Base para futuras modificaciones:** Los commits sirven como puntos de referencia estables para futuras modificaciones.

Pero concretamente ¿Qué es un commit?

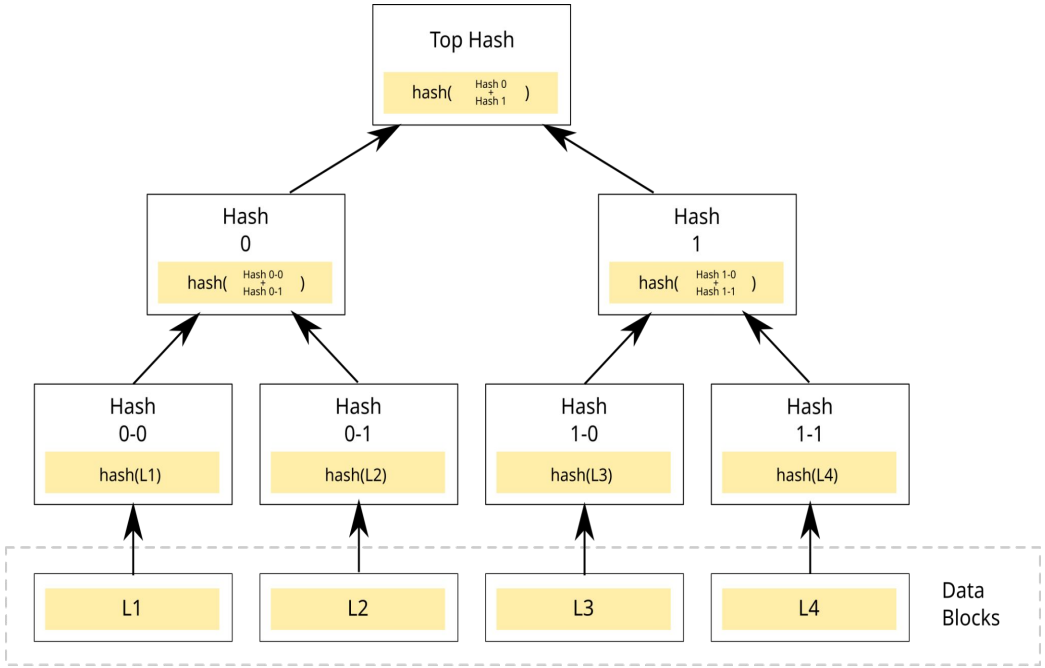
Es un hash criptográfico* de: Todos los archivos, Mensaje de commit, Autor, fecha, etc y commit padre.

Criptográfico = que no se puede invertir, ni encontrar colisiones (eficientemente).

Con optimizaciones para no re-computar el hash desde cero cada vez (ver Merkle trees).

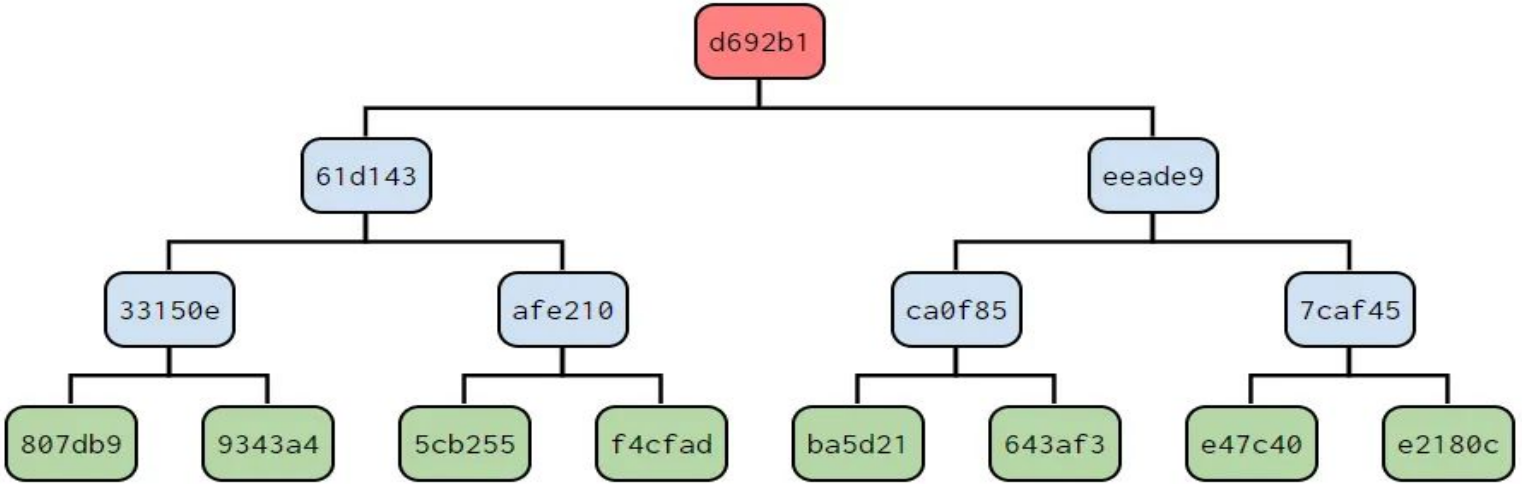
Merkle tree

Es un árbol en el que cada nodo hoja está etiquetado con un hash criptográfico de un bloque de datos, y cada nodo no hoja está etiquetado con un hash criptográfico de los labels de sus nodos hijos.

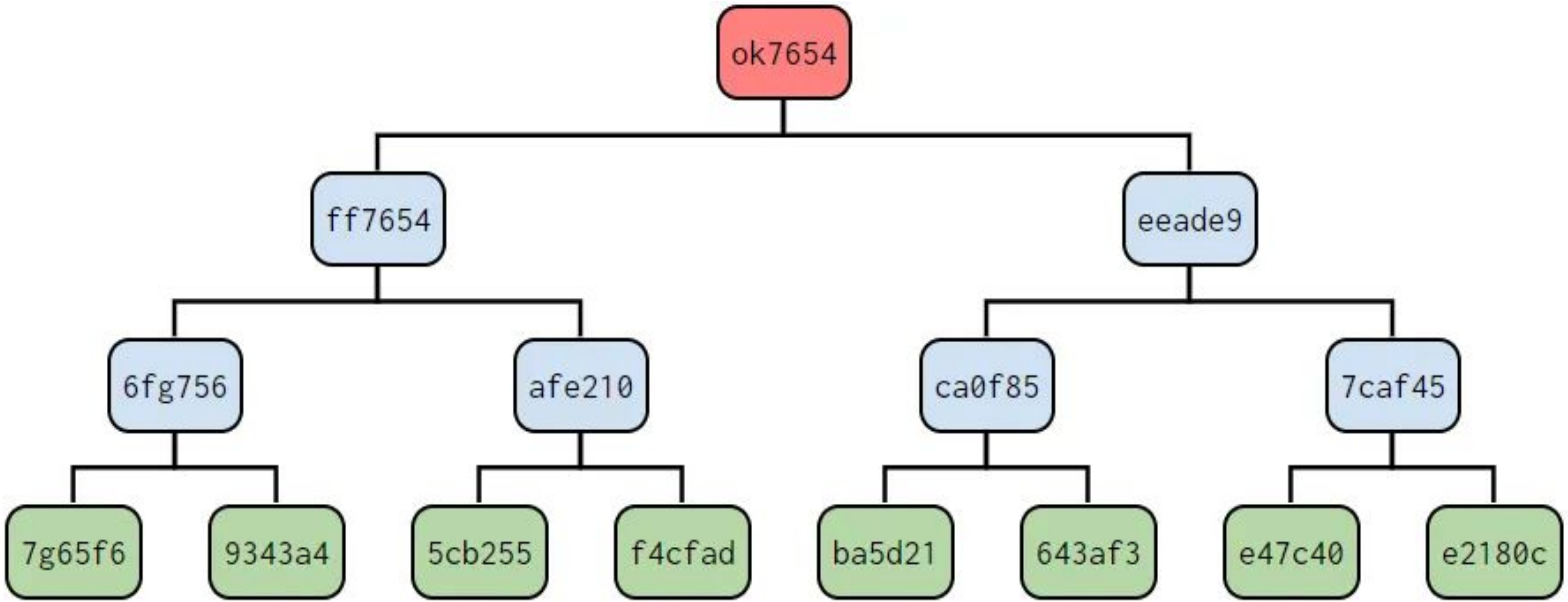


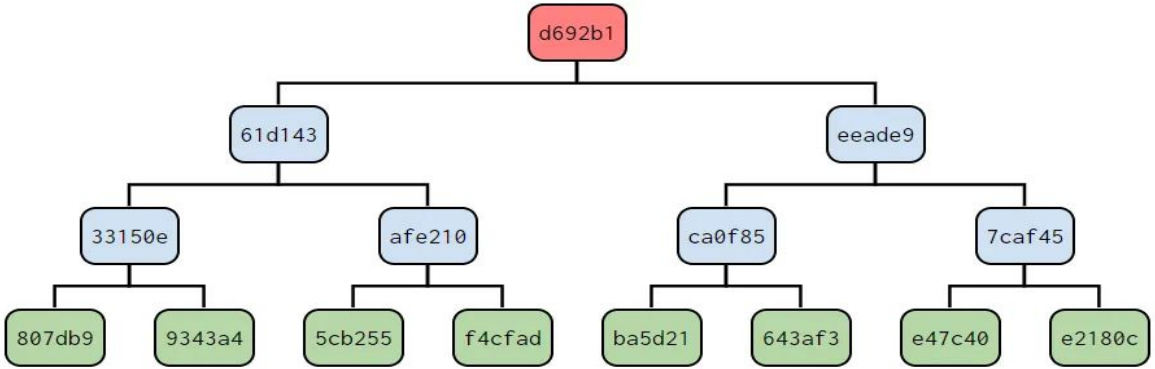
Un ejemplo,

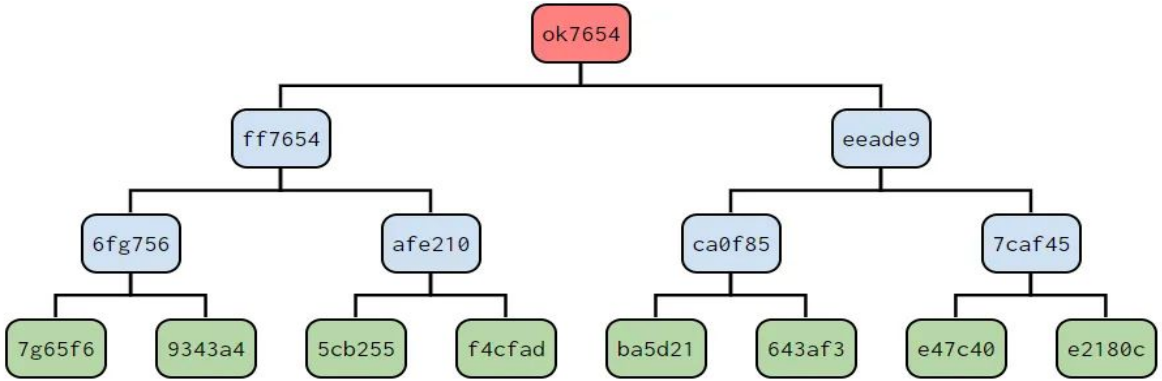
Supongamos que tenemos 8 archivos cuyo árbol de Merkle es



Cambiamos el primer archivo




d692b1
61d143
eeade9
33150e
afe210
ca0f85
7caf45
807db9
9343a4
5cb255
f4cfad
ba5d21
643af3
e47c40
e2180c


ok7654
ff7654
eeade9
6fg756
afe210
ca0f85
7caf45
7g65f6
9343a4
5cb255
f4cfad
ba5d21
643af3
e47c40
e2180c

Objetos de git (.git/objects)

Git almacena el contenido en objetos del tipo tree o blob. Un objeto tree contiene una o más entrada de un hash SHA-1 de un blob o un sub-tree el cual tiene asociado un modo, tipo y nombre de archivo

```
git cat-file -p master^{tree}

100644 blob 2c99a52776c2084e7b8d5852aa13429985db5d16    main.c
```

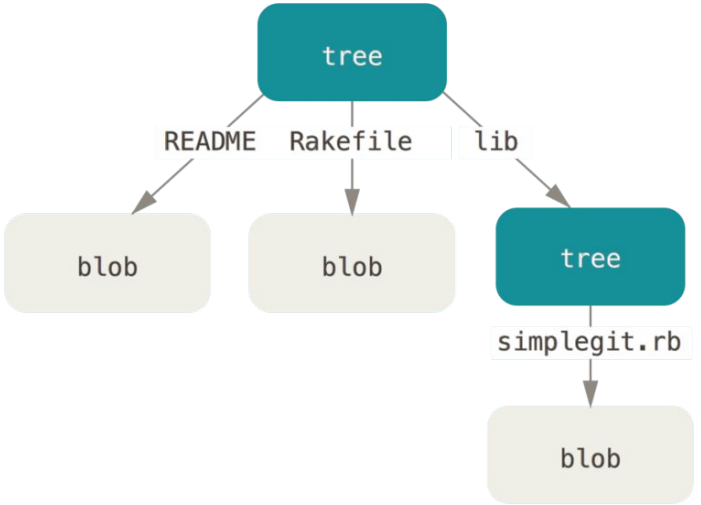
La sintaxis master^{tree} especifica que el objeto de tipo tree que es apuntado por el último commit en el branch master.

Los objetos blobs tienen un snapshot de los archivos del repositorio.

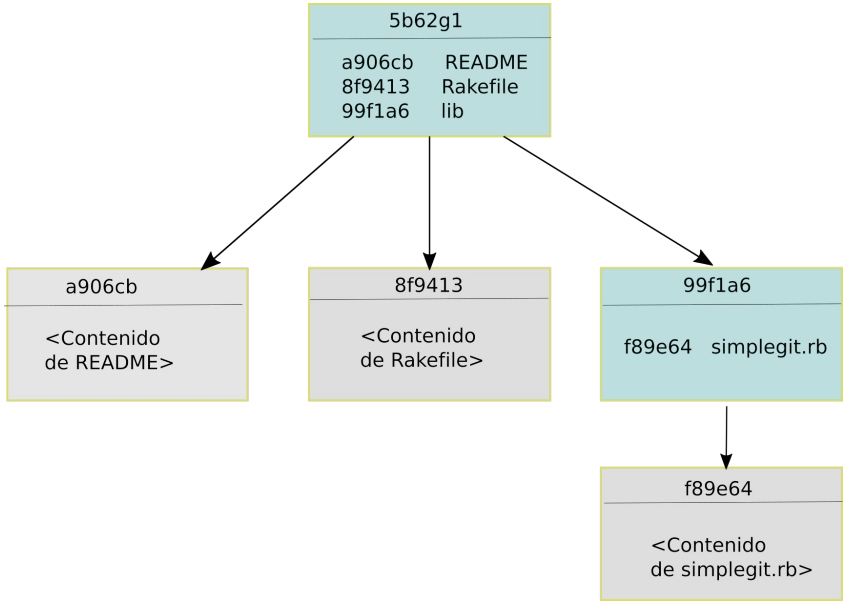
Ejemplo

Supongamos que tenemos un proyecto que tiene la siguiente estructura:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859    README
100644 blob 8f94139338f9404f26296befa88755fc2598c289    Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0    lib
```

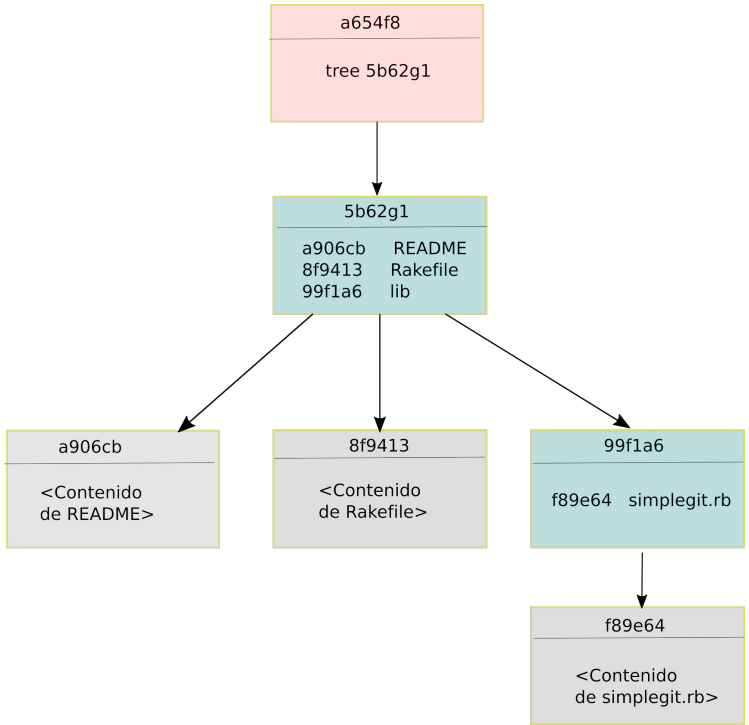


Los objetos tree, y los blobs asociados a sus claves



Cuando hacemos commit

Cuando hacemos un commit se crea un objeto de tipo tree.



Git Commit: Buenas prácticas

- Commit lo más pequeños posibles
- Nunca romper el build.
- Mensajes descriptivos: → Guías de Estilo

Estructura de mensaje:

Tipo: Subject

Cuerpo del mensaje

Tipo→ feat, doc, fix, refactor

https://udacity.github.io/git-styleguide/

https://github.com/agis/git-style-guide

Borrar archivos y config

git rm borra un archivo (y anota el cambio en la staging area). Es lo mismo que hacer rm y git add.

```
~/proyecto$ git rm main.c
```

git config configura opciones globales

```
git config --global user.name "Alan Turing"

$ git config --global user.email "aturing@princeton.edu"
```

Remotes

~/proyecto$ git remote add origin git@github.com:eevidal/proyecto.git

~/proyecto$ git push -u origin master

Enumerating objects: 9, done.

Counting objects: 100% (9/9), done.

Delta compression using up to 4 threads

Compressing objects: 100% (5/5), done.

Writing objects: 100% (9/9), 789 bytes | 789.00 KiB/s, done.

Total 9 (delta 0), reused 0 (delta 0), pack-reused 0

To github.com:eevidal/proyecto

* [new branch]

Resúmen: comandos básicos de Git

- `git init`: Inicializa un repositorio Git
- `git clone`: Clona un repositorio git
- `git add`: Agrega archivos al índice de Git
- `git rm`: Borra un archivo y anota el cambio en el staging area.
- `git commit`: Crea un nuevo commit
- `git push`: Envía los cambios al repositorio remoto
- `git pull`: Descarga los cambios del repositorio remoto

Documentación Oficial: https://git-scm.com/doc

Revirtiendo cambios

git checkout <file>: revierte cambios locales.

git reset: vacía el staging area.

git reset <commit>: vuelve al commit, sin modificar archivos.

git reset --hard <commit>: vuelve al commit, descartando todo.

Branches

- Un branch es un nombre que “apunta” o “sigue” a un commit hash
- git branch featureX: crear y moverse a un branch
- git checkout <b>: cambiar de branch
- master suele ser la rama principal

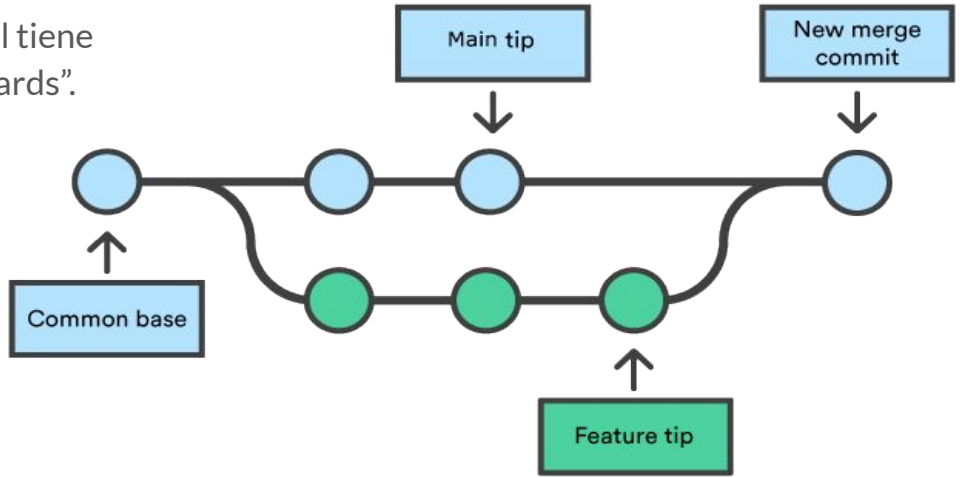
Merges

Toma dos commits y une sus cambios en uno.

• En el branch *source* git merge <commit>. Pull tiene merge implícito, push sólo permite “fast-forwards”.

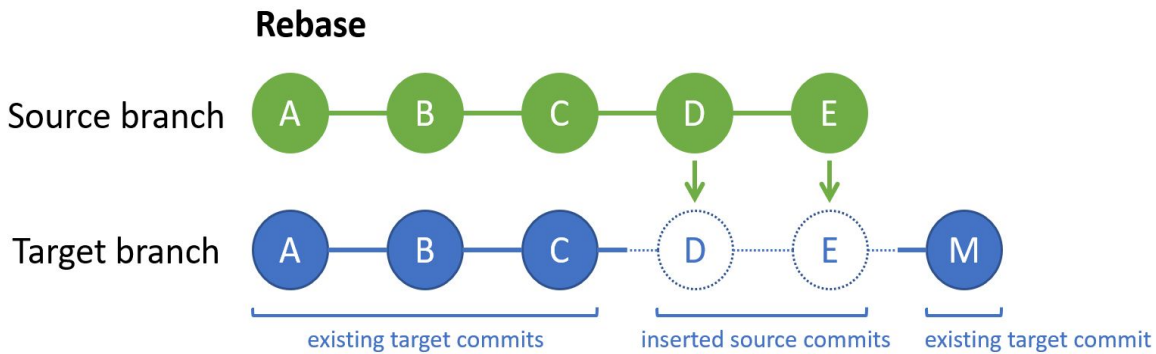
• Posiblemente haya que **corregir conflictos**

• Interfaz gráfica: gitg (o git log --graph)



Rebase

- Similar a un merge... pero “reescribe la historia”



Links útiles

https://wizardzines.com/comics/inside-git/

https://wizardzines.com/comics/every-git-command/

Referencias

- Libro: https://git-scm.com/book
- TryGit: https://try.github.io/
- A Visual Git Reference: https://marklodato.github.io/visual-git-guide/index-en.html
- Git From the Bottom Up: http://ftp.newartisans.com/pub/git.from.bottom.up.pdf
- Why Git is Better than X: https://bryankaraffa.github.io/whygitisbetter/
- Learn Git Branching: https://learngitbranching.js.org/
- Charla Torvalds: https://www.youtube.com/watch?v=4XpnKHJAok8
- Undertanding Merkle trees: https://medium.com/geekculture/understanding-merkle-trees-f48732772199