Introducción a Sistemas de Control de Versión y Git

Iván Arcuschin Moreno

DC - FCEyN - UBA

11/03/2016

Motivación 1/2

Trabajando solo

- Tenemos la versión final del TP en un archivo llamado `tp_version_final.pdf`

Motivación 1/2

Trabajando solo

- Tenemos la versión final del TP en un archivo llamado `tp_version_final.pdf`
- Nos damos cuenta de un cambio a último momento: `tp_version_final_2.pdf`

Motivación 1/2

Trabajando solo

- Tenemos la versión final del TP en un archivo llamado `tp_version_final.pdf`
- Nos damos cuenta de un cambio a último momento: `tp_version_final_2.pdf`
- Correcciones para la re-entrega: `tp_version_final_reentrega.pdf`

Motivación 1/2

Trabajando solo

- Tenemos la versión final del TP en un archivo llamado `tp_version_final.pdf`
- Nos damos cuenta de un cambio a último momento: `tp_version_final_2.pdf`
- Correcciones para la re-entrega: `tp_version_final_reentrega.pdf`
- Cambios a la re-entrega: `tp_version_final_reentrega_2.pdf`

Motivación 1/2

Trabajando solo

- Tenemos la versión final del TP en un archivo llamado `tp_version_final.pdf`
- Nos damos cuenta de un cambio a último momento: `tp_version_final_2.pdf`
- Correcciones para la re-entrega: `tp_version_final_reentrega.pdf`
- Cambios a la re-entrega: `tp_version_final_reentrega_2.pdf`

Motivación 2/2

Trabajando de a grupo

- Enviar cambios por mail, o

Motivación 2/2

Trabajando de a grupo

- Enviar cambios por mail, o
- Sincronizar cambios por Dropbox, o

Motivación 2/2

Trabajando de a grupo

- Enviar cambios por mail, o
- Sincronizar cambios por Dropbox, o
- Sincronizar cambios por Google Docs.

Motivación 2/2

Trabajando de a grupo

- Enviar cambios por mail, o
- Sincronizar cambios por Dropbox, o
- Sincronizar cambios por Google Docs.

Historia 1/2

El núcleo de Linux es un proyecto de software de código abierto con un alcance bastante grande[1]:

- Más de 15 **millones** de lineas de código en su versión *3.2*
- En promedio, cada día se añaden 3.500 lineas de código.
- Más de 1300 desarrolladores involucrados en la versión *3.2*

Durante la mayor parte del mantenimiento del núcleo de Linux (1991-2002), los cambios en el software se pasaron en forma de parches y archivos.

En 2002, el proyecto del núcleo de Linux empezó a usar un Sistema de Control de Versión propietario llamado BitKeeper.

[1] Fuente: http://royal.pingdom.com/2012/04/16/linux-kernel-development-numbers/

Historia 2/2

En 2005, la relación entre la comunidad que desarrollaba el núcleo de Linux y la compañía que desarrollaba BitKeeper se vino abajo, y la herramienta dejó de ser ofrecida gratuitamente.

Esto impulsó a la comunidad de desarrollo de Linux (y en particular a Linus Torvalds, el creador de Linux) a desarrollar su propia herramienta basada en algunas de las lecciones que aprendieron durante el uso de BitKeeper.

Algunos de los objetivos del nuevo sistema fueron los siguientes:

- Velocidad
- Diseño sencillo
- Fuerte apoyo al desarrollo no lineal (miles de ramas paralelas)
- Completamente distribuido
- Capaz de manejar grandes proyectos (como el núcleo de Linux) de manera eficiente

Desde su nacimiento en 2005, Git ha evolucionado y madurado para ser fácil de usar y aún conservar estas cualidades iniciales.

Instalación

Linux

- En Debian (ej: Ubuntu): `sudo apt-get install git-all`
- Otros: http://git-scm.com/download

Windows

Git Bash: https://git-for-windows.github.io/

Otros

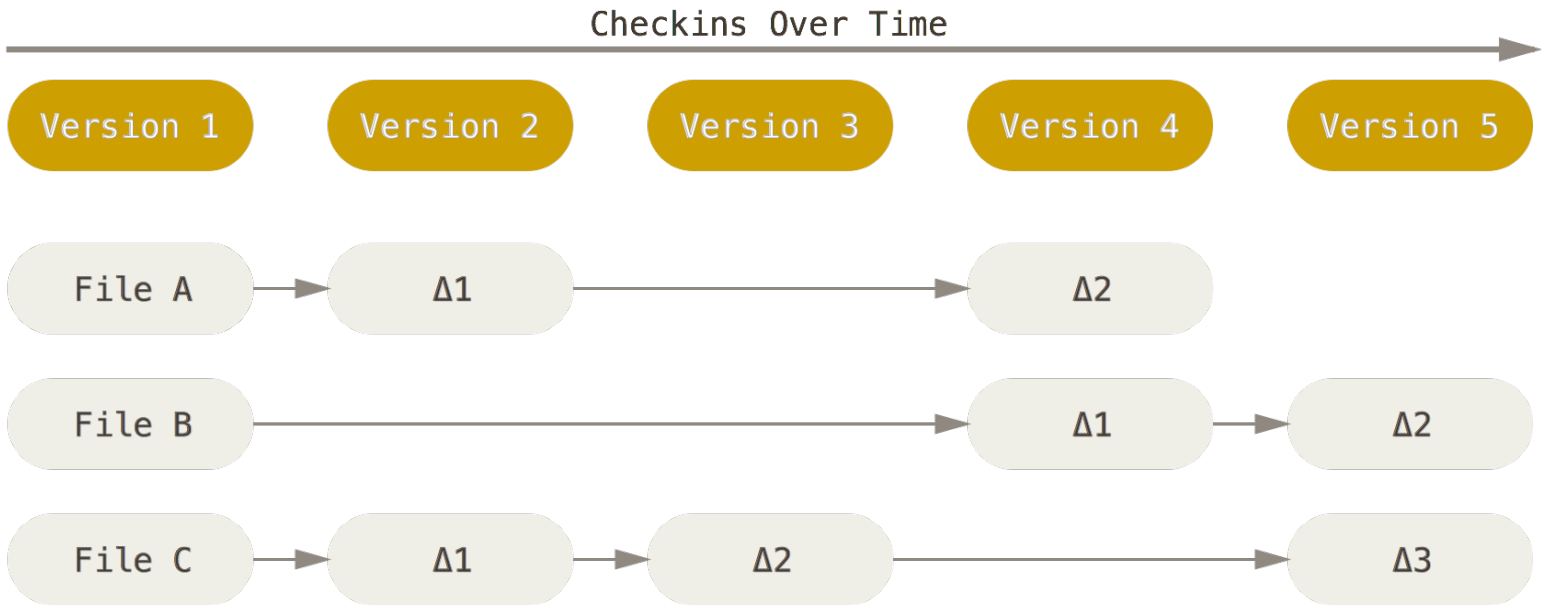
Getting started installing Git:
https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

Clientes

GUI Clients: http://git-scm.com/downloads/guis

Git vs Subversion y otros 1/2

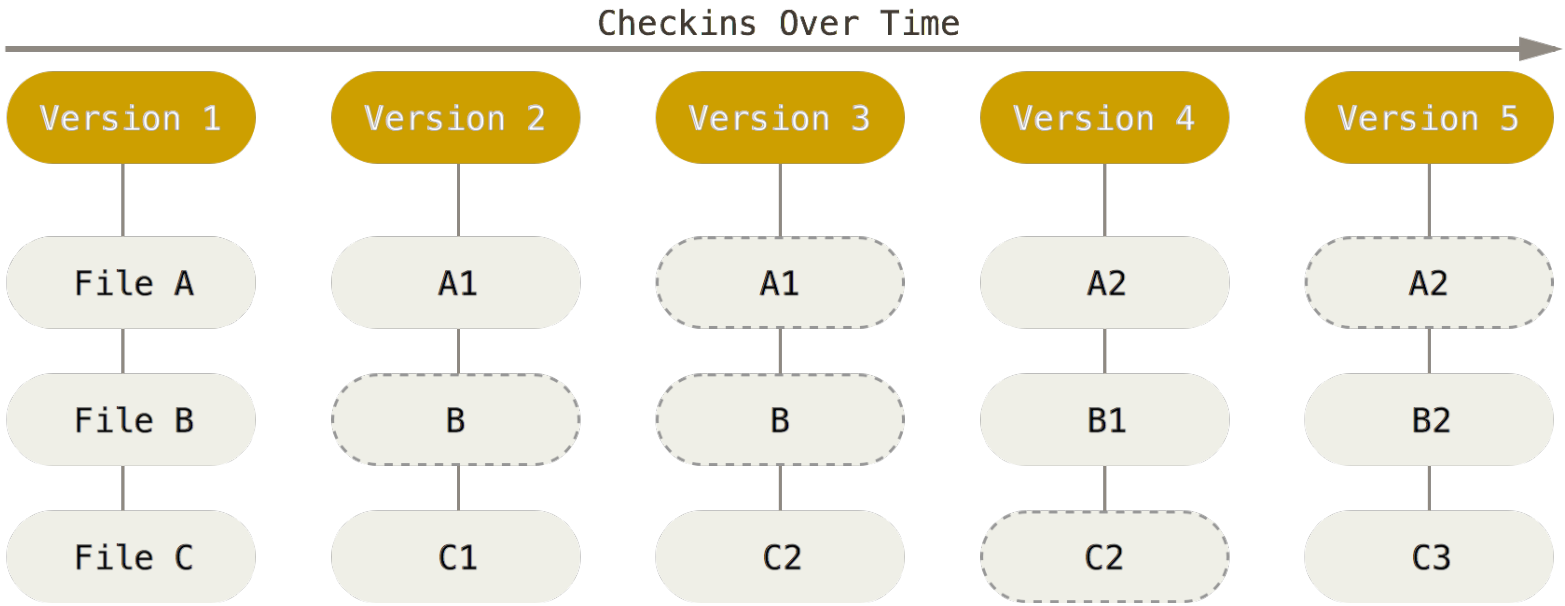
Conceptualmente, la mayoría de los demás Sistemas de Control de Versión (CVS, Subversion, Perforce, Bazaar, etc.) almacenan la información como una lista de cambios en los archivos.



Git vs Subversion y otros 2/2

En cambio, Git modela sus datos más como un conjunto de snapshots de un mini sistema de archivos.

Cada vez que confirmas un cambio, se crea un snapshot del aspecto de todos tus archivos en ese momento, y se guarda una referencia. Para ser eficiente, si los archivos no se han modificado, Git no almacena el archivo de nuevo, sólo un link al archivo anterior idéntico que ya tiene almacenado.



Configurando Git por primera vez (git config)

Tu identidad

Es importante establecer nuestro nombre y email ya que estos van asociados con los cambios de manera inmutable:

```
git config --global user.name "John Doe"
git config --global user.email johndoe@example.com
```

Editor de texto

Para editar el mensaje asociado a un cambio, se utiliza un editor de texto. Por defecto es Vi o Vim, pero podemos cambiarlo:

```
git config --global core.editor emacs
```

Output con colores

```
git config color.ui true
```

Obteniendo un repositorio Git

Inicializando un repositorio en un directorio existente

Para crear un nuevo repositorio local podemos ir al directorio y ejecutar `git init`. Esto crea un subdirectorio *.git* que tiene todos los archivos necesarios del repositorio.

Clonando un repositorio existente

Para obtener una copia local de un repositorio existente en algún servidor, utilizamos el comando `git clone [url]` (En Subversion es `svn checkout [url]`).
Por ejemplo:

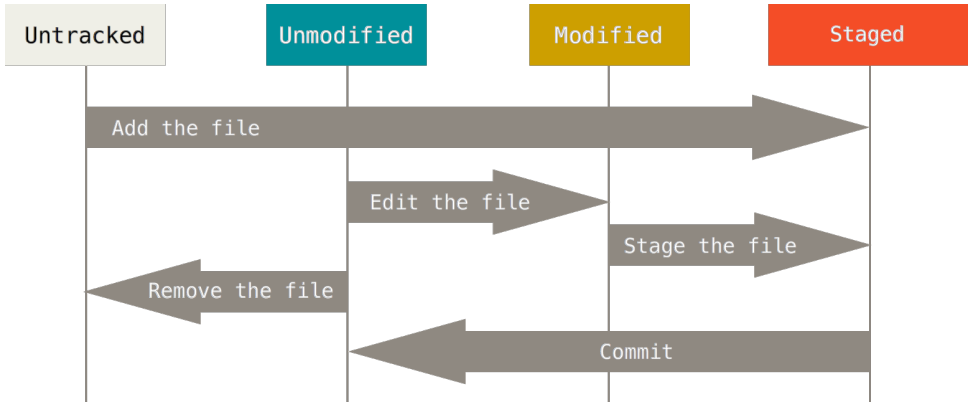
```
git clone git@github.com:FlyingPumba/git-intro.git
```

Clona un repositorio que tiene los fuentes de estas diapositivas !

Comprobando el estado de tus archivos

El comando que utilizamos para ver el estado actual de los archivos es `git status`. Git tiene tres estados principales en los que se pueden encontrar los archivos:

- **Modificado (modified)**: significa que has modificado el archivo pero todavía no lo has confirmado a tu base de datos.
- **Preparado (staged)**: significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.
- **Confirmado (committed)**: significa que los datos están almacenados de manera segura en tu base de datos local.



Haciendo cambios - Agregando un archivo

Supongamos que acabamos de agregar el archivo `README`. Si corremos el comando `git status` nos dirá algo como:

```
Untracked files:
   README
nothing added to commit but untracked files present
```

Esto significa básicamente que el archivo `README` no se encuentra en la última snapshot que tiene Git del repositorio.
Para comenzar el seguimiento de este archivo utilizamos el comando `git add`. En nuestro ejemplo: `git add README`. Luego, al volver a correr `git status`, veremos que:

```
Changes to be committed:
   new file:   README
```

Haciendo cambios - Modificando un archivo

Supongamos que ahora modificamos el archivo `README`. Si corremos el comando `git status` nos dirá algo como:

```
Changes not staged for commit:
   modified:   README
```

Lo que significa que el archivo `README` ha sido modificado desde la última snapshot que tiene Git del repositorio.
Para pasar estas modificaciones a *Preparado (staged)*, también usamos `git add`. De vuelta: `git add README`. Y al correr `git status`, veremos que:

```
Changes to be committed:
   modified:   README
```

Viendo los cambios preparados

Muchas veces es conveniente ver cuales son los cambios realizados antes de mandarlos a Preparado. Para esto tenemos el comando `git diff`, que muestra las diferencias entre lo que hay en tu directorio de trabajo con lo que hay en tu área de preparación.

Si en cambio, queremos ver los cambios que preparamos y que irán en la próxima confirmación, podemos usar `git diff --staged`.

Confirmando ("commiteando") cambios

Una vez que tenemos nuestros cambios listos en el area de preparación, podemos utilizar el comando `git commit` para confirmarlos. Ningún cambio o archivo que no esté en el area de preparación se incluirá en la confirmación.

Cada confirmación requiere de un mensaje con una descripción de los cambios realizados. Por ejemplo: `git commit -m ''Agrego archivo README''`. Si ejecutamos `git commit` sin el parámetro `-m`, se nos abrirá el editor de texto configurado para Git, en el que podremos escribir una descripción de forma más cómoda.

Viendo la historia de los commits

Después de haber hecho varios *commits*, o si acabamos de clonar un repositorio de internet, puede que querramos ver el historial de commits para ver cuales fueron las modificaciones que se hicieron. Esto se puede lograr utilizando el comando `git log`.

Trabajando con repositorios remotos

Los repositorios remotos son versiones de tu proyecto que se encuentran alojados en Internet o en algún punto de la red. Puedes tener varios, cada uno de los cuales puede ser de sólo lectura, o de lectura/escritura, según los permisos que tengas.

Colaborar con otros implica gestionar estos repositorios remotos, y mandar (push) y recibir (pull) datos de ellos cuando necesites compartir cosas.

Para ver qué repositorios remotos están configurados, podemos usar el comando `git remote`. Si acabamos de clonar el repositorio que tiene los fuentes de estas diapositivas y corremos `git remote -v`, veremos:

```
origin  git@github.com:FlyingPumba/git-intro.git (fetch)
origin  git@github.com:FlyingPumba/git-intro.git (push)
```

Y si tenemos un repositorio creado localmente y queremos añadir un remoto, usamos `git remote add [nombre] [url]`.

Ramificaciones en Git

Cada vez que realizamos una confirmación (commit), Git almacena entre otras cosas uno o varios punteros a las confirmaciones que sean padres directos de esta.
Una rama o branch en Git no es más que un puntero especial parado en un commit. Ejecutar `git branch` nos mostrará las ramas actuales. Veamos un ejemplo sencillo:

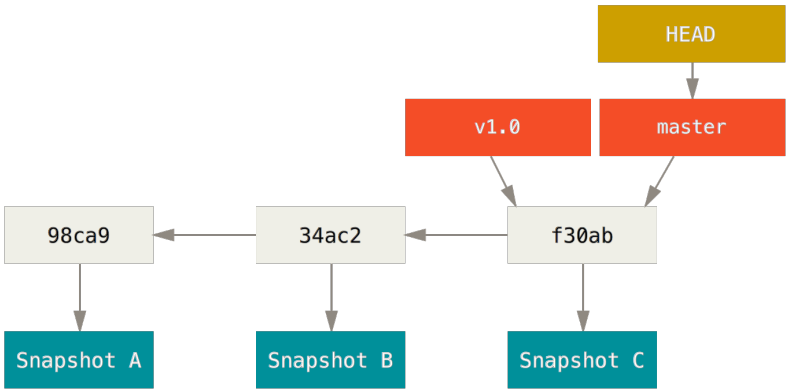


Figura 1: La rama "master" apunta al commit *f30ab*

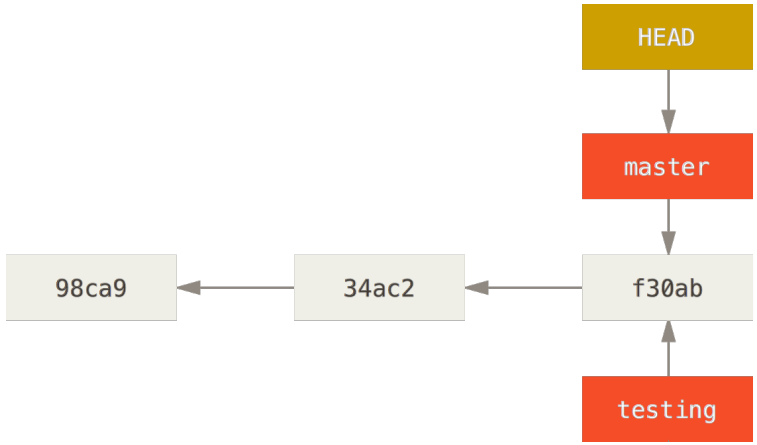


Figura 2: Creamos una nueva rama ejecutando `git branch testing`

Cambiando de rama

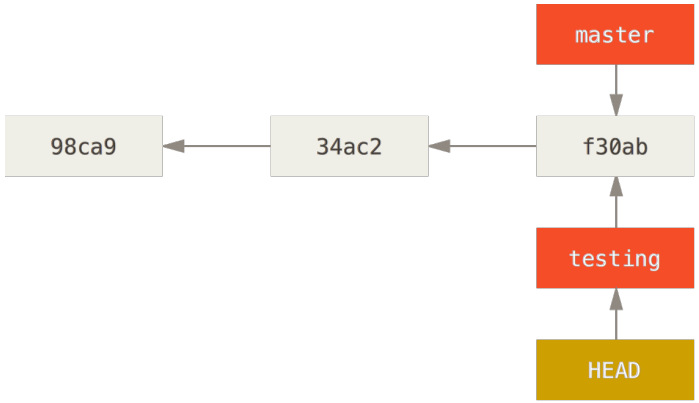


Figura 3: Si queremos cambiar de rama, podemos ejecutar `git checkout testing`

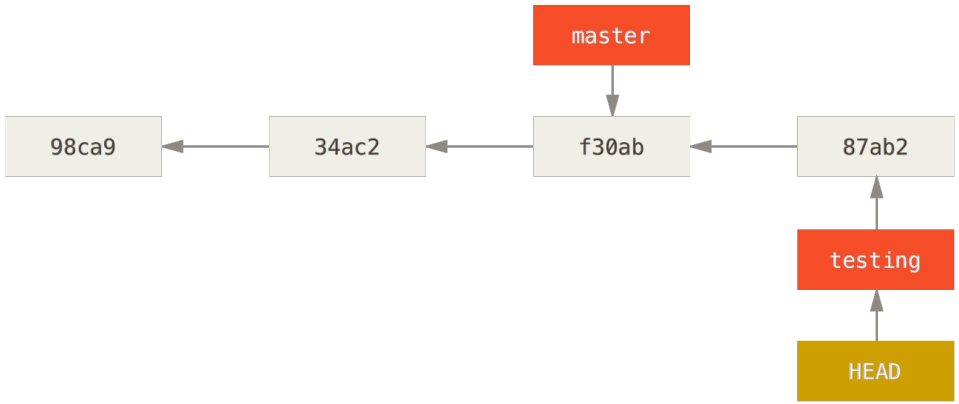


Figura 4: Y los siguientes commits serán agregados a la rama "testing"

Ramas remotas

Ademas de tener ramas locales, podemos tenemos ramas remotas, que referencian al estado de ramas en tus repositorios remotos. Si por ejemplo quisieramos saltar a la rama "master" en el repositorio remoto "origin" podríamos utilizar el comando `git checkout origin/master`.

Nuestras ramas locales no sincronizan automaticamente con los repositorios remotos. Por esta razón, si tenemos una rama llamada "master" de forma local que apunta a la rama remota "origin/master" podemos actualizarla parandonos en "master" y ejecutando `git pull origin master`.

De forma análoga, si queremos publicar cambios locales aun repositorio remoto, tenemos el comando `git push`. Si estamos parados en la rama "master": `git push origin master`.

Fusionando ramas

Supongamos que tenemos una rama "iss53" en la cual hemos terminado de arreglar un error, y queremos combinarla con la rama "master". Para ello, simplemente nos paramos en "master" (`git checkout master`) y ejecutamos: `git merge iss53`.

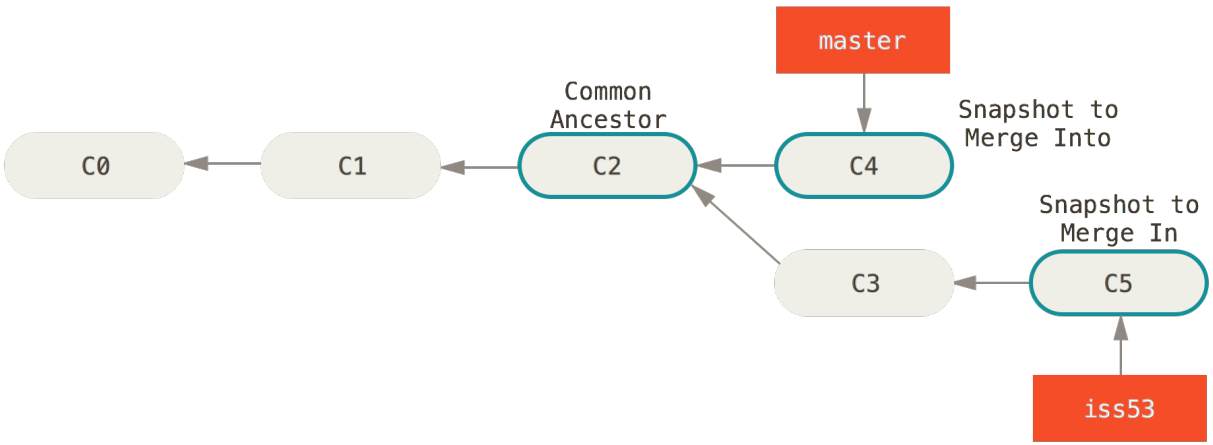


Figura 5: Antes del merge

Fusionando ramas

Supongamos que tenemos una rama "iss53" en la cual hemos terminado de arreglar un error, y queremos combinarla con la rama "master". Para ello, simplemente nos paramos en "master" (`git checkout master`) y ejecutamos: `git merge iss53`.

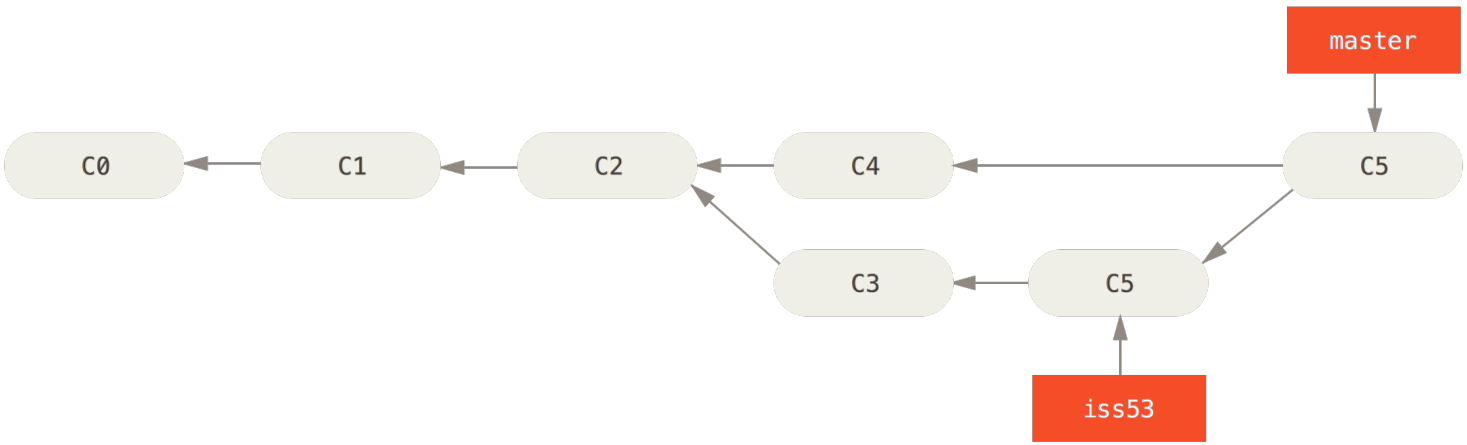


Figura 6: Después del merge

Otros comandos útiles

- `git fetch [remote repository]`: para traer todos los datos de un repositorio remoto.
- `git reset`: para deshacer cambios, ya sea en el area de trabajo o en los commits.
- `git rm`: para borrar un archivo y pasarlo al area de preparación.
- `git mv`: para mover/renombrar un archivo y pasarlo al area de preparación.
- `git show`: muestra información de distintos objectos de Git: commits, tags, etc.
- `git stash`: guarda el estado del area de trabajo y lo limpia.
- `git rebase [branch]`: aplica todos los commits que difieren entre un branch y el que estamos parados.
- `git tag [etiqueta] [commit hash]`: ponerle una etiqueta a un commit.

Bibliografía extra

- Git Community book, disponible online y en español: https://git-scm.com/book/es/v2
- `git help [command]` para ver la documentación de cualquier comando de Git.
- A visual Git reference: http://marklodato.github.io/visual-git-guide/index-es.html
- Try Git online: https://try.github.io
- Git - SVN Crash Course: https://git-scm.com/course/svn.html

Mini ejercicio de práctica

Básico

1. Instalar Git en tu máquina.
2. Clonar el repositorio que tiene los fuentes de estas diapositivas. (git@github.com:FlyingPumba/git-intro.git)
3. Abrir el archivo *main.tex* y **arregkar** el typo en esta oración.
4. Pasar *main.tex* al area de preparados.
5. Confirmar los cambios.

Mini ejercicio de práctica

Avanzado

1. Crearse una cuenta en `GitHub.com`
2. Ir a la página del repositorio que tiene los fuentes de estas diapositivas `https://github.com/FlyingPumba/git-intro` y *forkear* el proyecto a un repositorio personal en GitHub.
3. Agregar a su repositorio local un nuevo remoto llamado "fork" que apunte a su repositorio remoto en GitHub.
4. Pushear la rama "master" al repositorio remoto "fork".
5. Hacer un *Pull request* en GitHub.