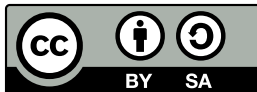


Introducción a Git y Github para desarrollo colaborativo

Antonio García Domínguez

University of York



27 de marzo de 2016

Contenidos

- 1 Introducción
- 2 Trabajo local
- 3 Trabajo distribuido

Materiales en <http://github.com/bluezio/curso-git>.

Contenidos

- 1 **Introducción**
 - Antecedentes
 - Tipos de SCV
- 2 Trabajo local
- 3 Trabajo distribuido

Contenidos

- 1 **Introducción**
 - Antecedentes
 - Tipos de SCV
- 2 Trabajo local
- 3 Trabajo distribuido

¿Por qué usar un SCV?

Copiar ficheros y mandar correos no escala

- ¿Cuál era la última versión?
- ¿Cómo vuelvo a la anterior?
- ¿Cómo reúno mis cambios con los de otro?

SCV: todo ventajas a cambio de alguna disciplina

- Llevamos un historial de los cambios
- Podemos ir trabajando en varias cosas a la vez
- Podemos colaborar con otros
- Hacemos copia de seguridad de todo el historial

Historia de los SCV

Sin red, un desarrollador

1972 Source Code Control System

1980 Revision Control System

Centralizados

1986 Concurrent Version System

1999 Subversion («CVS done right»)

Distribuidos

2001 Arch, monotone

2002 Darcs

2005 Git, Mercurial (hg), Bazaar (bzt)

Historia de Git

Antes de BitKeeper

Para desarrollar Linux, se usaban parches y tar.gz.

BitKeeper

02/2002 BitMover regala licencia BitKeeper (privativo)

04/2005 BitMover retira la licencia tras roces

Git

04/2005 Linus Torvalds presenta Git, que ya reúne ramas

06/2005 Git se usa para gestionar Linux

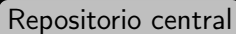
02/2007 Git 1.5.0 es utilizable por mortales

03/2016 Última versión: Git 2.7.4

Contenidos

- 1 **Introducción**
 - Antecedentes
 - Tipos de SCV
- 2 Trabajo local
- 3 Trabajo distribuido

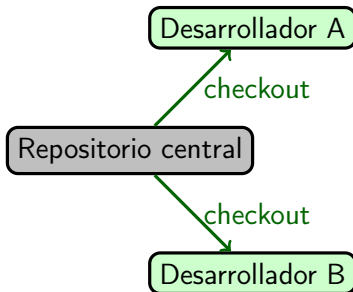
SCV centralizados



Repositorio central

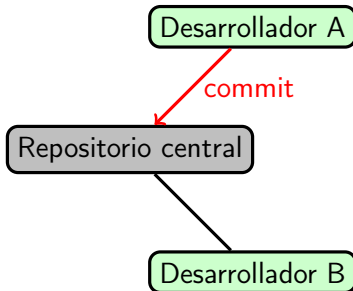
Tenemos nuestro repositorio central con todo dentro.

SCV centralizados



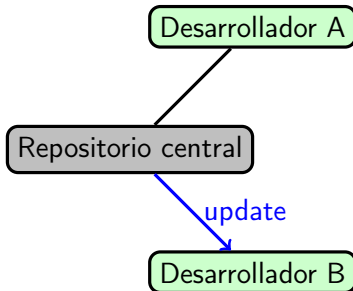
Los desarrolladores crean **copias de trabajo**.

SCV centralizados



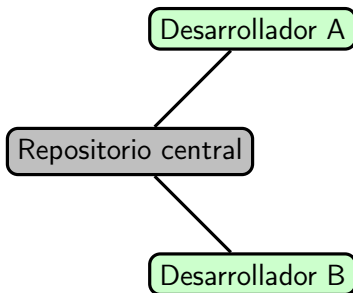
El desarrollador A manda sus cambios al servidor.

SCV centralizados



El desarrollador B los recibe.

SCV centralizados



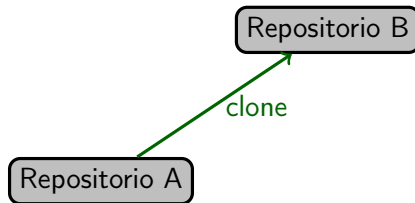
¿Y si se cae el servidor, o la red?

SCV distribuidos

Repositorio A

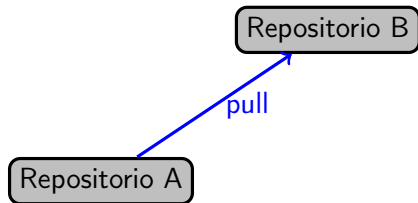
Tenemos nuestro repositorio.

SCV distribuidos



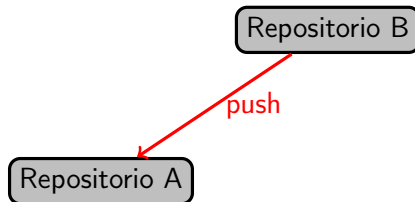
Alguien **clona** el repositorio.

SCV distribuidos



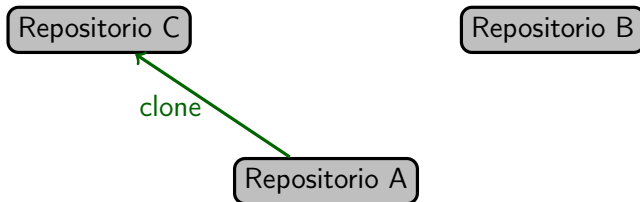
De vez en cuando se trae nuestros cambios recientes.

SCV distribuidos



De vez en cuando nos manda sus cambios.

SCV distribuidos



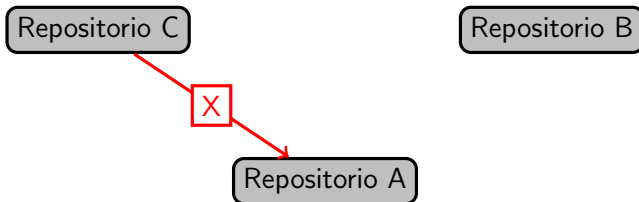
Viene otro desarrollador.

SCV distribuidos



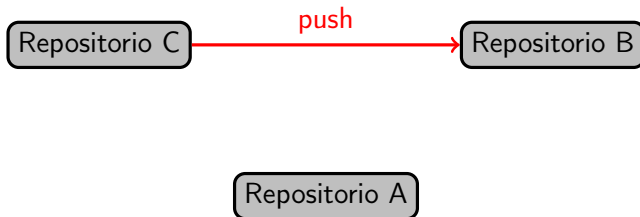
Intenta hacer sus cambios locales...

SCV distribuidos



Pero no le funciona, o no tiene permisos para ello.

SCV distribuidos



Se los pasa al otro desarrollador sin más.

SCV distribuidos



La diferencia entre los repositorios es *social*, no técnica.

Ventajas de un SCV distribuido (I)

Rapidez

- Todo se hace en local: el disco duro es más rápido que la red, y cuando esté todo en caché será más rápido aún
- Clonar un repositorio Git suele tardar *menos* que crear una copia de trabajo de SVN, y ocupa menos

Revisiones pequeñas y sin molestar

- Nadie ve nada nuestro hasta que lo mandamos
- Podemos ir haciendo revisiones pequeñas intermedias
- Sólo mandamos cuando compila y supera las pruebas
- Podemos hacer experimentos de usar y tirar

Ventajas de un SCV distribuido (I)

Rapidez

- Todo se hace en local: el disco duro es más rápido que la red, y cuando esté todo en caché será más rápido aún
- Clonar un repositorio Git suele tardar *menos* que crear una copia de trabajo de SVN, y ocupa menos

Revisiones pequeñas y sin molestar

- Nadie ve nada nuestro hasta que lo mandamos
- Podemos ir haciendo revisiones pequeñas intermedias
- Sólo mandamos cuando compila y supera las pruebas
- Podemos hacer experimentos de usar y tirar

Ventajas de un SCV distribuido (II)

Trabajo sin conexión

- En el tren, avión, autobús, etc.
- Aunque no tengamos permisos de escritura
- Aunque se caiga la red, se puede colaborar

Robustez

Falla el disco duro del repositorio bendito. ¿Qué hacer?

- Centralizado: copias de seguridad
- Distribuido: copias de seguridad y/o colaborar por otros medios

Ventajas de un SCV distribuido (II)

Trabajo sin conexión

- En el tren, avión, autobús, etc.
- Aunque no tengamos permisos de escritura
- Aunque se caiga la red, se puede colaborar

Robustez

Falla el disco duro del repositorio bendito. ¿Qué hacer?

- Centralizado: copias de seguridad
- Distribuido: copias de seguridad y/o colaborar por otros medios

Cuándo NO usar Git

Git no escala ante muchos ficheros binarios

- No sirve para llevar las fotos
- Ni para almacenar vídeos

Git no guarda metadatos

No sirve como sistema de copias de seguridad

Contenidos

- 1 Introducción
- 2 Trabajo local
 - Preparación
 - Conceptos
 - Operaciones comunes
- 3 Trabajo distribuido

Contenidos

- 1 Introducción
- 2 Trabajo local
 - Preparación
 - Conceptos
 - Operaciones comunes
- 3 Trabajo distribuido

Instalación de Git

Ubuntu Linux

- Mínimo: instalar git
- Recomendado: instalar gitk y tkdiff

Windows

- Original: <http://msysgit.github.io/>
- TortoiseGit: <https://tortoisegit.org/>
- Otros: SmartGit (\$79/persona), SourceTree...

Configuración inicial

Cambiamos la configuración global en *\$HOME/.gitconfig*

Identificación

```
$ git config --global user.name "Mi Nombre"  
$ git config --global user.email mi@correo
```

Editor: por defecto Vi/Vim

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

Herramienta para resolver conflictos

```
$ git config --global merge.tool tkdiff
```

Algunos alias útiles

```
$ git config --global alias.ci commit  
$ git config --global alias.st status  
$ git config --global alias.ai "add -i"
```

Creación de un repositorio

Sólo tenemos que ir a un directorio y decirle a Git que cree un repositorio ahí.

```
$ mkdir ejemplo
$ cd ejemplo
$ git init
Initialised empty Git repository in
/home/antonio/Documents/curso-git/presentaciones/esi-2016/ejemplo/.git
/
```


Nuestras dos primeras revisiones en la rama master

```
$ echo "hola" > f.txt
$ git add f.txt
$ git commit -m "primer commit"
[master (root-commit) 3533601] primer commit
 1 file changed, 1 insertion(+)
 create mode 100644 f.txt
$ echo "adios" >> f.txt
$ git add f.txt
$ git commit -m "segundo commit"
[master dfc46ed] segundo commit
 1 file changed, 1 insertion(+)
```

- ¿Qué es ese identificador después de «root-commit»?
- ¿Dónde se guardan mis cosas?
- ¿git add dos veces?

Contenidos

- 1 Introducción
- 2 Trabajo local
 - Preparación
 - **Conceptos**
 - Operaciones comunes
- 3 Trabajo distribuido

Modelo de datos de Git

Características

- Un repositorio es un grafo orientado acíclico de objetos
- Hay 4 tipos de objetos: *commit*, *tree*, *blob* y *tag*
- Los objetos son direccionables por contenido (resumen SHA1)

Consecuencias del diseño

- Los objetos son inmutables: al cambiar su contenido, cambia su SHA1
- Git *no* gestiona información de ficheros movidos y demás
- Git *nunca* guarda más de un objeto una vez en el DAG, aunque aparezca en muchos sitios

Modelo de datos de Git: revisiones (*commits*)

Contenido

- Fecha, hora, autoría, fuente y un mensaje
- Referencia a revisión padre y a un *tree*

```
$ git cat-file -p HEAD
tree 65de8c1fce51aedbc5b0c838d5d2be0883b3ab0e
parent 3533601d4f6b9859bd67b97500deb1282cd07160
author Antonio <a@b.com> 1459098997 +0200
committer Antonio <a@b.com> 1459098997 +0200
```

segundo commit

Modelo de datos de Git: árboles (*trees*)

Contenido

- Lista de *blobs* y *trees*
- Separa el nombre de un fichero/directorio de su contenido
- Sólo gestiona los bits de ejecución de los ficheros
- No se guardan directorios vacíos

```
$ git cat-file -p HEAD:  
100644 blob 9114647dde3052c36811e94668f951f623d8005d f.txt
```

Modelo de datos de Git: ficheros (*blobs*)

Contenido

Secuencias de bytes sin ningún significado particular.

```
$ git cat-file -p HEAD:f.txt  
hola  
adios
```

Modelo de datos de Git: etiquetas (*tags*)

Contenido

- Referencias simbólicas inmutables a otros objetos
- Normalmente apuntan a *commits*
- Pueden firmarse mediante GnuPG, protegiendo la integridad de todo el historial hasta entonces

```
$ git tag -a v1.0 -m "version 1.0" HEAD
$ git cat-file -p v1.0
object dfc46edc7db71c6c7dd3ee6d6f2fd2c886565598
type commit
tag v1.0
tagger Antonio <a@b.com> 1459098998 +0200

version 1.0
```

Estructura física de un repositorio Git

Partes de un repositorio Git

- Directorio de trabajo
- Grafo de objetos: *.git*
- Área de preparación: *.git/index*

```
$ ls .git  
branches      config      HEAD      index      logs      refs  
COMMIT_EDITMSG  description hooks      info      objects
```


Área de preparación, caché o índice

Concepto

Instantánea que vamos construyendo de la siguiente revisión.

Diferencia entre Git y otros SCV

- `svn add` = añadir fichero a control de versiones
- `git add` = añadir contenido a área de preparación

Consecuencias

- Controlamos exactamente qué va en cada revisión
- Algo raro hasta acostumbrarse, pero es *muy* potente

Preparando revisiones (I)

```
$ echo "bueno" >> f.txt  
$ git add f.txt  
$ echo "malo" >> f.txt  
$ echo "nuevo" > g.txt
```

Preparando revisiones (II)

```
$ git status
```

```
On branch master
```

```
Changes to be committed:
```

```
(use "git reset HEAD <file>..." to unstage)
```

```
modified:   f.txt
```

```
Changes not staged for commit:
```

```
(use "git add <file>..." to update what will be committed)
```

```
(use "git checkout -- <file>..." to discard changes in working  
directory)
```

```
modified:   f.txt
```

```
Untracked files:
```

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

```
g.txt
```

Preparando revisiones (III)

```
$ git diff --staged
diff --git a/f.txt b/f.txt
index 9114647..3d0a14e 100644
--- a/f.txt
+++ b/f.txt
@@ -1,2 +1,3 @@
 hola
 adios
+bueno
```

```
$ git diff
diff --git a/f.txt b/f.txt
index 3d0a14e..ad3ec81 100644
--- a/f.txt
+++ b/f.txt
@@ -1,3 +1,4 @@
 hola
 adios
 bueno
+mallo
```

Preparando revisiones (IV)

```
$ git commit -m "tercer commit"
[master 1dcf1eb] tercer commit
 1 file changed, 1 insertion(+)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)
```

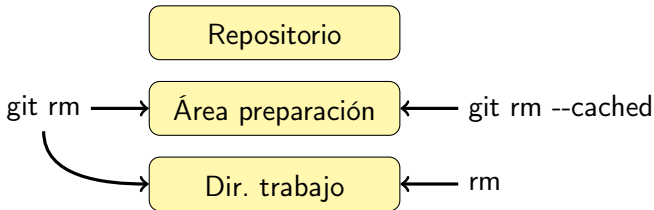
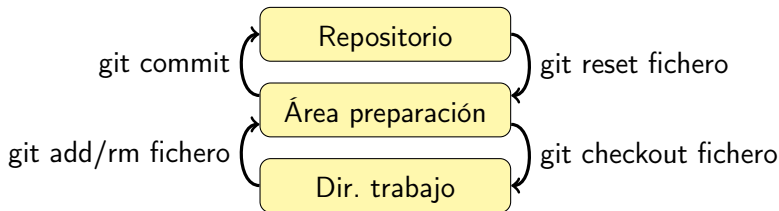
```
modified:   f.txt
```

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

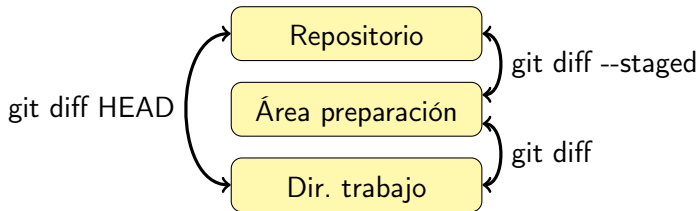
```
g.txt
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

Esquemas de órdenes para añadir y eliminar



Esquema de órdenes para comparar



Contenidos

- 1 Introducción
- 2 Trabajo local
 - Preparación
 - Conceptos
 - Operaciones comunes
- 3 Trabajo distribuido

Historial: por línea de órdenes (I)

```
$ git log
commit 1dcf1ebc0511e110b8a7c65e027ff0fd6ea9e92b
Author: Antonio <a@b.com>
Date:   Sun Mar 27 19:16:38 2016 +0200
```

tercer commit

```
commit dfc46edc7db71c6c7dd3ee6d6f2fd2c886565598
Author: Antonio <a@b.com>
Date:   Sun Mar 27 19:16:37 2016 +0200
```

segundo commit

```
commit 3533601d4f6b9859bd67b97500deb1282cd07160
Author: Antonio <a@b.com>
Date:   Sun Mar 27 19:16:37 2016 +0200
```

primer commit

Historial: por línea de órdenes (II)

```
$ git log --pretty=oneline  
1dcf1ebc0511e110b8a7c65e027ff0fd6ea9e92b tercer commit  
dfc46edc7db71c6c7dd3ee6d6f2fd2c886565598 segundo commit  
3533601d4f6b9859bd67b97500deb1282cd07160 primer commit
```

Historial: por línea de órdenes (III)

```
$ git log --graph --pretty=oneline --decorate=short --abbrev-commit
* 7eec99d Transparencias: movido descripción del repositorio m...
* 8c90bb1 transparencias.tex: configure UI coloring during rep...
* b40fa05 Remove cmd.tmp after running the command, to avoid i...
* b5cb1b3 Transparencias: comenzado a trabajar en nuevas trans...
* 1f74aee Añadido PDF del taller de la Quincena de la Ingenier...
* c550afd Merge branch 'spanish' of gitorious.org:spanish-gi...
| \
| * 4d536a3 (origin/spanish-git-reflect...
* | c7859e7 Añadido guión de instalación de Git
* | 8d33923 Merge branch 'spanish' of gitorious.org:...
| \ \
| | /
| * 2769854 Advanced topics: forgot some text at the l...
* db25cd9 Añadido guión para instalar Git
```

Historial: más cosas

Opciones útiles

- Por autor: `--author`, `--committer`
- Por fecha: `--since`, `--until`
- Por cambio: `-S`
- Por mensaje: `--grep`
- Con parches: `-p` (resumidos: `--stat`)

Otras órdenes

- `git show`: una revisión determinada
- `git whatchanged`: estilo SVN
- `gitk`: interfaz gráfica

Ayuda

Listado de órdenes

- `git help` da un listado breve
- `git help --all` las lista todas (144+)
- Muchas son «fontanería»: sólo usamos la «porcelana»

Sobre una orden concreta

- `man git-orden`
- `git help orden`
- `git help -w orden` (en navegador)
- `git orden -h`

Contenidos

1 Introducción

2 Trabajo local

3 Trabajo distribuido

- Manejo de ramas
- Interacción con repositorios remotos

Contenidos

- 1 Introducción
- 2 Trabajo local
- 3 Trabajo distribuido
 - Manejo de ramas
 - Interacción con repositorios remotos

Clonar un repositorio

Métodos de acceso

- git://: anónimo, de sólo lectura
- SSH: siempre cifrado y con autenticación
- HTTP(S): se lleva bien con cortafuegos

```
$ git clone https://neptuno.uca.es/git/sandbox-git  
Cloning into 'sandbox-git'...
```


Ramas en Git

Concepto: líneas de desarrollo

`master` Rama principal

`develop` Rama de desarrollo

`nueva-cosa` Rama para añadir algo concreto («feature branch»)

Diferencias con otros SCV

- No son apañes con directorios, sino parte del modelo de datos
- Rama en Git: referencia mutable y compartible a una revisión
- Etiqueta en Git: referencia *inmutable* a un objeto

Listando las ramas

Ramas locales

```
$ git branch  
* master
```

Ramas remotas

```
$ git branch -r  
origin/HEAD -> origin/master  
origin/ejemplo-conflicto  
origin/ejemplo-heurísticas  
origin/ejemplo-merge-ff  
origin/ejemplo-merge-master  
origin/ejemplo-merge-noff  
origin/ejemplo-rebase-i  
origin/master
```

Gestionando ramas

Crear ramas

```
$ git branch mirama HEAD
$ git branch
* master
  mirama
```

Borrar ramas

```
$ git branch -d mirama
Deleted branch mirama (was ef24155).
$ git branch -d master
error: Cannot delete the branch 'master' which you are currently on.
$ git branch
* master
```

Cambiando entre ramas

A una rama local

No confundir con `git checkout -- master`, que copia el fichero *master* del índice al directorio de trabajo.

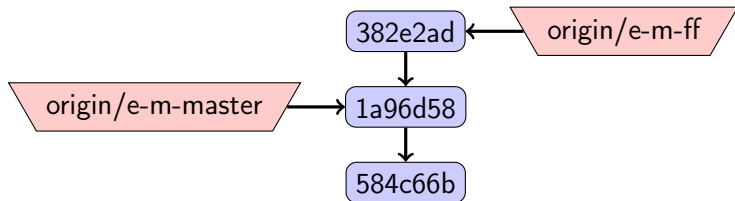
```
$ git checkout master
Already on 'master'
Your branch is up-to-date with 'origin/master'.
```

A una rama remota

Es de sólo lectura: creamos una rama local que la siga.

```
$ git checkout -b ejemplo-merge-ff origin/ejemplo-merge-ff
Switched to a new branch 'ejemplo-merge-ff'
Branch ejemplo-merge-ff set up to track remote branch
ejemplo-merge-ff from origin.
```

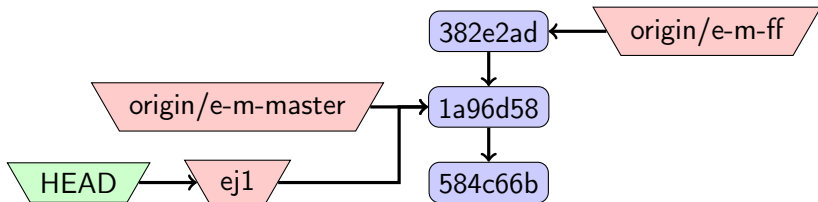
Reuniendo ramas: «fast-forward»



Podemos comprobar cómo están las ramas con:

```
$ gitk origin/ejemplo-merge-master origin/ejemplo-merge-ff
```

Reuniendo ramas: «fast-forward»



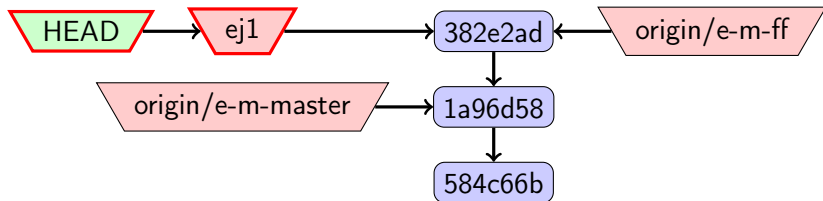
Vamos a crear la rama desde la que haremos la reunión:

```
$ git checkout -b ej1 origin/ejemplo-merge-master
```

Switched to a new branch 'ej1'

Branch ej1 set up to track remote branch ejemplo-merge-master from origin.

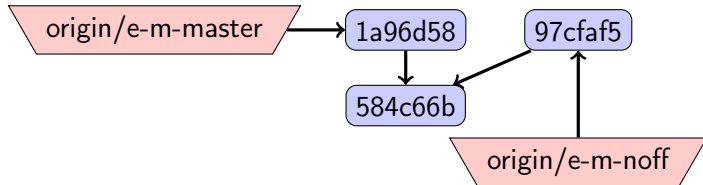
Reuniendo ramas: «fast-forward»



La reunión consiste en adelantar la referencia sin más:

```
$ git merge origin/ejemplo-merge-ff
Updating 1a96d58..382e2ad
Fast-forward
 hola_mundo.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

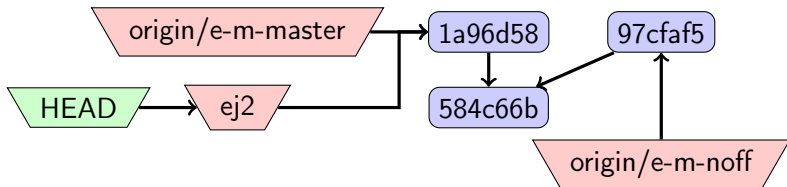
Reuniendo ramas: «recursive»



Otra forma de ver las ramas es con:

```
$ git log --graph --decorate \  
origin/ejemplo-merge-master origin/ejemplo-merge-noff
```


Reuniendo ramas: «recursive»



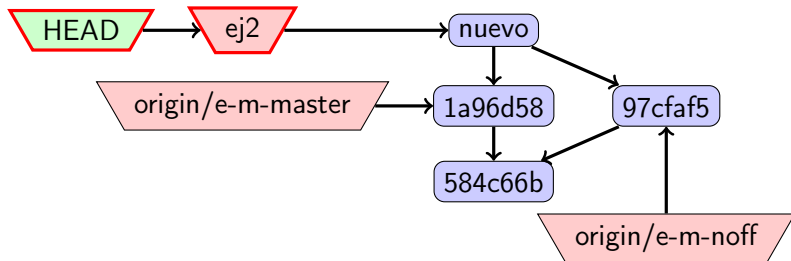
Creamos otra vez una rama nueva para el punto de partida:

```
$ git checkout -b ej2 origin/ejemplo-merge-master
```

Switched to a new branch 'ej2'

Branch ej2 set up to track remote branch ejemplo-merge-master from origin.

Reuniendo ramas: «recursive»



La reunión tiene que crear una nueva revisión con dos padres:

```
$ git merge origin/ejemplo-merge-noff
Auto-merging hola_mundo.c
Merge made by the 'recursive' strategy.
 hola_mundo.c | 4 +++++
 1 file changed, 4 insertions(+)
```

Reuniendo ramas: conflicto (I)

Miremos el historial de dos ramas, a ver qué cambian:

```
$ gitk origin/ejemplo-merge-master origin/ejemplo-conflicto
```

Vamos a intentar reunir las:

```
$ git checkout -b ej3 origin/ejemplo-merge-master
```

```
Switched to a new branch 'ej3'
```

```
Branch ej3 set up to track remote branch ejemplo-merge-master from  
origin.
```

```
$ git merge origin/ejemplo-conflicto
```

```
Auto-merging hola_mundo.c
```

```
CONFLICT (content): Merge conflict in hola_mundo.c
```

```
Automatic merge failed; fix conflicts and then commit the result.
```

Reuniendo ramas: conflicto (II)

Si es texto: `git mergetool`

Lanza herramienta gráfica para resolver todos los conflictos.

Si es binario o no nos gusta `git mergetool` (!)

- `git checkout --ours`: nos quedamos con lo que teníamos
- `git checkout --theirs`: nos quedamos con lo que reunimos
- Editamos los marcadores a mano (como en SVN)
- Después preparamos con `git add` y creamos la revisión con `git commit`, dejando la información acerca del conflicto resuelto en el mensaje

Contenidos

- 1 Introducción
- 2 Trabajo local
- 3 Trabajo distribuido**
 - Manejo de ramas
 - Interacción con repositorios remotos**

Envío de objetos

En general: `git push URL origen:destino`

- URL se puede reemplazar por apodo (`origin`)
- origen es rama o etiqueta local
- destino es rama o etiqueta remota
- Actualiza destino en rep. remoto a origen
- Sólo tiene éxito si es un «fast-forward»

Observaciones

- `git push URL x = git push URL x:x`
- `git push URL` actualiza todas las ramas remotas que se llamen igual que las locales
- `git push` es `git push origin`

Recepción de objetos

```
$ git fetch --all  
Fetching origin
```

Efecto

Esta orden recibe todos los objetos nuevos de todos los repositorios remotos que conozcamos.

Nota

- Actualiza las ramas remotas
- Seguramente nos interesará traernos sus cambios con `git merge` después
- Como es muy típico, `git pull` combina las dos órdenes: `git fetch` seguido de `git merge`

Flujo de trabajo centralizado

Secuencia típica: muy similar a SVN

- Creamos un clon del repositorio *dorado*
- Nos actualizamos con `git pull`
- Si hay conflictos, los resolvemos
- Hacemos nuestros cambios sin preocuparnos mucho de Git
- Los convertimos en revisiones cohesivas y pequeñas
- Los enviamos con `git push`

Flujos de trabajo distribuidos: variantes

Un integrador

- Cada desarrollador tiene rep. privado y rep. público
- Los desarrolladores colaboran entre sí
- El integrador accede a sus rep. públicos y actualiza el repositorio oficial
- Del repositorio oficial salen los binarios
- Los desarrolladores se actualizan periódicamente al oficial

Director y tenientes

- El integrador (dictador) es un cuello de botella
- Se ponen intermediarios dedicados a un subsistema (teniente)

Forjas con alojamiento Git

La más popular: Github (<http://github.com>)

- Gratis para proyectos libres, de pago para proyectos cerrados
- Integra aspectos sociales y nuevas funcionalidades

Otra opción: Bitbucket (<http://bitbucket.org>)

- Gratis para proyectos libres, o cerrados hasta 5 colaboradores
- De pago para el resto

Alojamiento propio

- Motor de <http://repo.or.cz>
- Gitlab: <https://about.gitlab.com/>

Otras órdenes disponibles

- `bisect`: búsqueda binaria de defectos
- `blame`: autoría por líneas
- `bundle`: colaborar sin red
- `clean`: retirar ficheros fuera de control de versiones
- `format-patch`: preparar parches para enviar por correo
- `grep`: buscar subcadenas por el repositorio
- `instaweb`: visor Web
- `rebase -i`: navaja suiza para reorganizar ramas
- `rebase`: replantar ramas en base a otras
- `stash`: aparcacar cambios en zona temporal
- `svn`: interoperar con SVN
- `submodule`: incluir repositorios externos

¡Gracias por su atención!

antonio.garcia-dominguez@york.ac.uk
@antoniogado, @antoniogado_es