Sistemas de control de versiones distribuidos

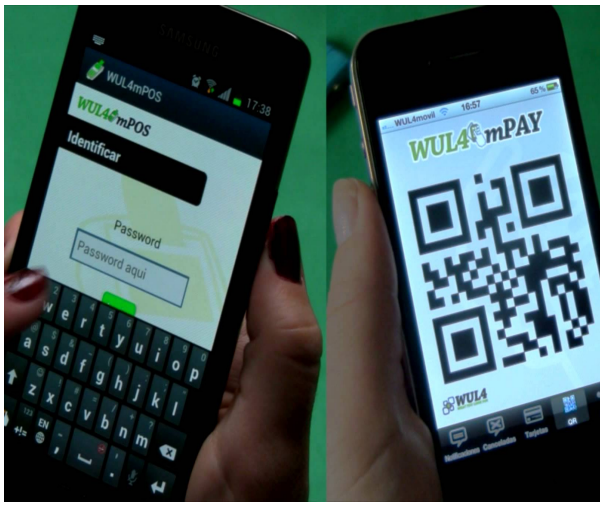
Controla las versiones de tu trabajo con GIT

Nacho Álvarez
@neonigmacdb
neonigma@gmail.com

16 de enero de 2014

Who?

- **Trayectoria profesional:** soporte UCO, desarrollador Web, desarrollador / integrador distribuciones GNU/Linux.
- **Actualmente:** WUL4 Córdoba (mobile + backend developer)
- **Involucrado en:**



Redsýs

Definiciones

- **Control de versiones:** gestión de los diversos cambios que se realizan sobre los elementos de algún producto
- Una **versión**, **revisión** o **edición** de un producto, es el **estado** en el que se encuentra dicho producto en un **momento** dado de su desarrollo o modificación
- Los **sistemas** de control de versiones (SCV) vienen a automatizar parcialmente la gestión de este control de cambios
- Existen SCV **centralizados** (repositorio único remoto) y SCV **distribuidos** (cada usuario su repositorio local + remoto)
- Los más famosos y de más trayectoria son: SVN, GIT, Mercurial, Bazaar, ClearCase, Perforce...

Ventajas SCV centralizados

- En los sistemas distribuidos hay un **mayor bloqueo** del estado final del proyecto que en los sistemas centralizados.
- En los sistemas centralizados las versiones vienen identificadas por un **número de versión**. En lugar de eso, en los sistemas distribuidos, cada versión tiene un identificador (hash) al que se le puede asociar una etiqueta (tag).
- La **curva de aprendizaje** es sensiblemente menor que en los sistemas distribuidos
- Requiere **menor intervención** del mantenedor

Ventajas SCV distribuidos

- Necesita **menos operaciones en red** => mayor autonomía y una mayor rapidez.
- Aunque se **caiga** el **repositorio remoto** la gente puede seguir trabajando
- Alta probabilidad de **reconstrucción** en caso de falla debido a su arquitectura distribuida
- Permite mantener **repositorios centrales más limpios**, el mantenedor decide
- El **servidor remoto** requiere **menos recursos** que los que necesitaría un servidor centralizado ya que gran parte del trabajo lo realizan los **repositorios locales**.
- Al ser los sistemas distribuidos **más recientes** que los sistemas centralizados, y al tener **más flexibilidad** por tener un repositorio local y otro/s remotos, estos sistemas han sido diseñados para hacer fácil el uso de **ramas locales y remotas** (creación, evolución y fusión) y poder aprovechar al máximo su potencial.

¿Por qué GIT?

¿Por qué GIT?


github
SOCIAL CODING
bitbucket
by ATLASSIAN
ORACLE
PostgreSQL
SAMSUNG
spring
Google
Telefónica
Telefónica I+D

En números

bitbucket by ATLASSIAN

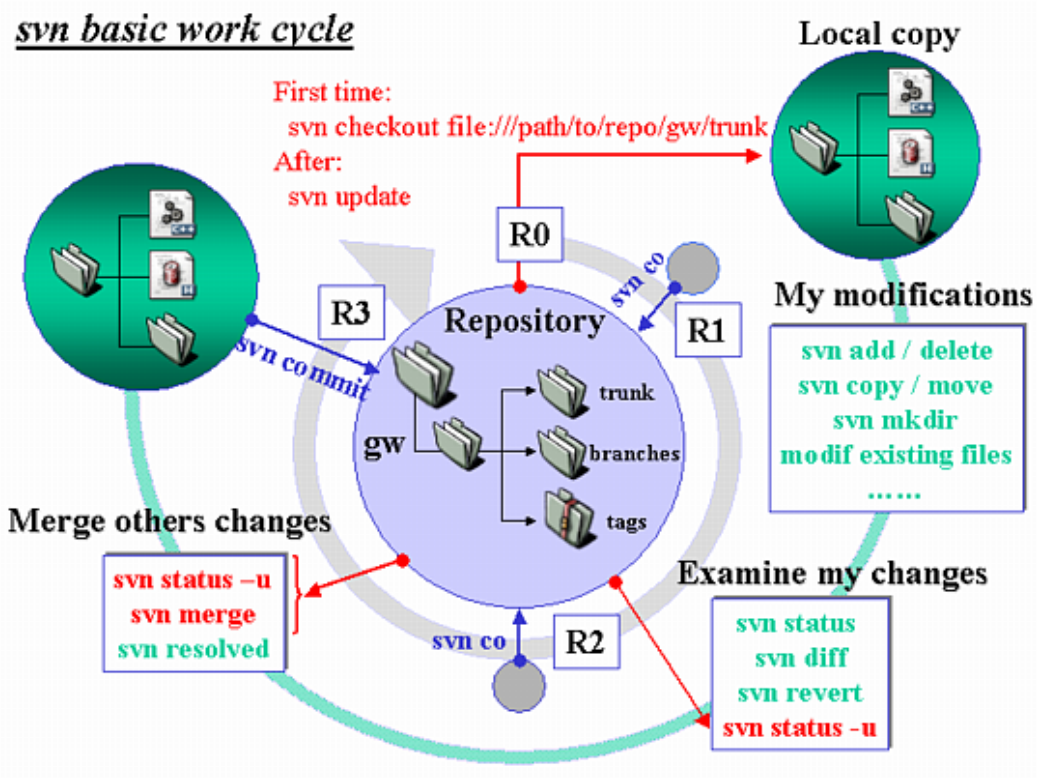
- Enfocado a **código privado** (Facebook, Adobe, Opera, WUL4...)
- Más de **1 millón** de usuarios registrados
- Integración del resto del ecosistema software: Bamboo (CI), Confluence (Doc), Jira (project tracking), SourceTree (GUI)...
- Se cobra por **número de integrantes de equipo**: 0-5 (gratis), 6-10 ($10 month), 11-25 ($25 month), 26-50 ($50 month)...

github SOCIAL CODING

- Enfocado a **código abierto** (bootstrap, nodejs, jquery, Amazon...)
- Más de **5 millones** de usuarios registrados y **10 millones** de **repositorios** creados
- Se cobra por **repositorios privados**: 5 ($7 month), 10 ($12 month), 20 ($22 month), 50 ($50 month)

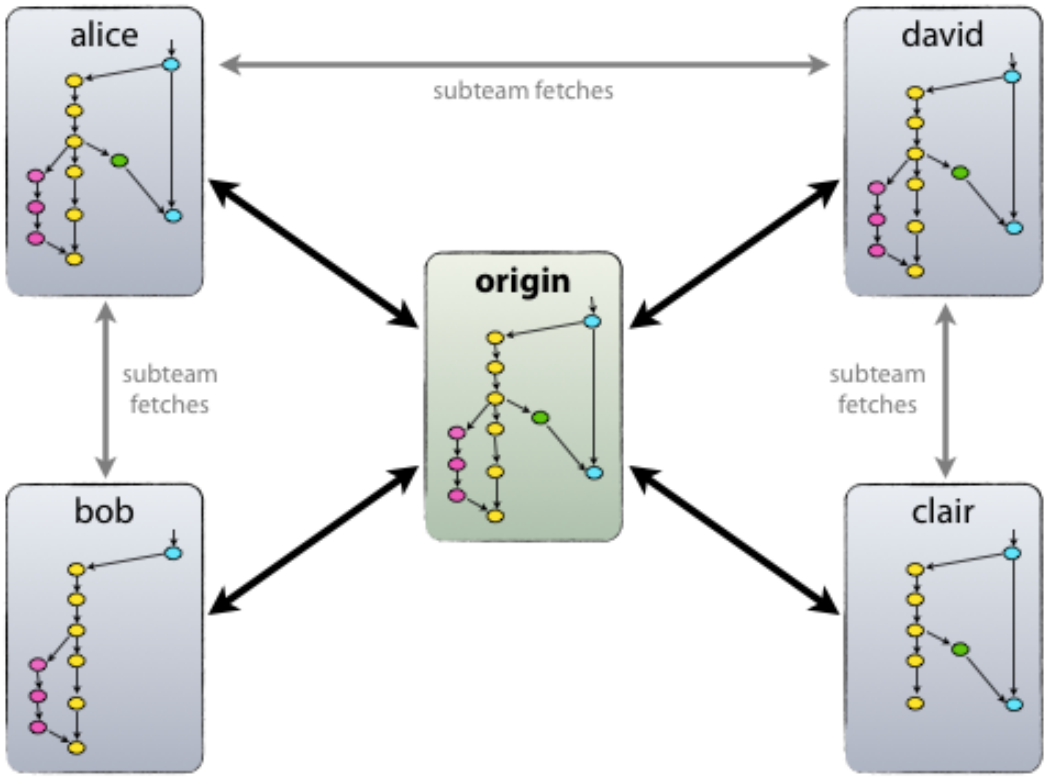
Arquitectura SCV centralizado

- Todo el mundo actualiza un mismo repositorio central remoto
- La gestión de ramas y tags es más una convención que parte de la herramienta



Arquitectura SCV distribuido

- Todo el mundo mantiene su copia del proyecto
- Cada integrante del equipo trabaja en sus propias funcionalidades en su repositorio local particular
- El mantenedor del repositorio acepta o no las modificaciones de los integrantes

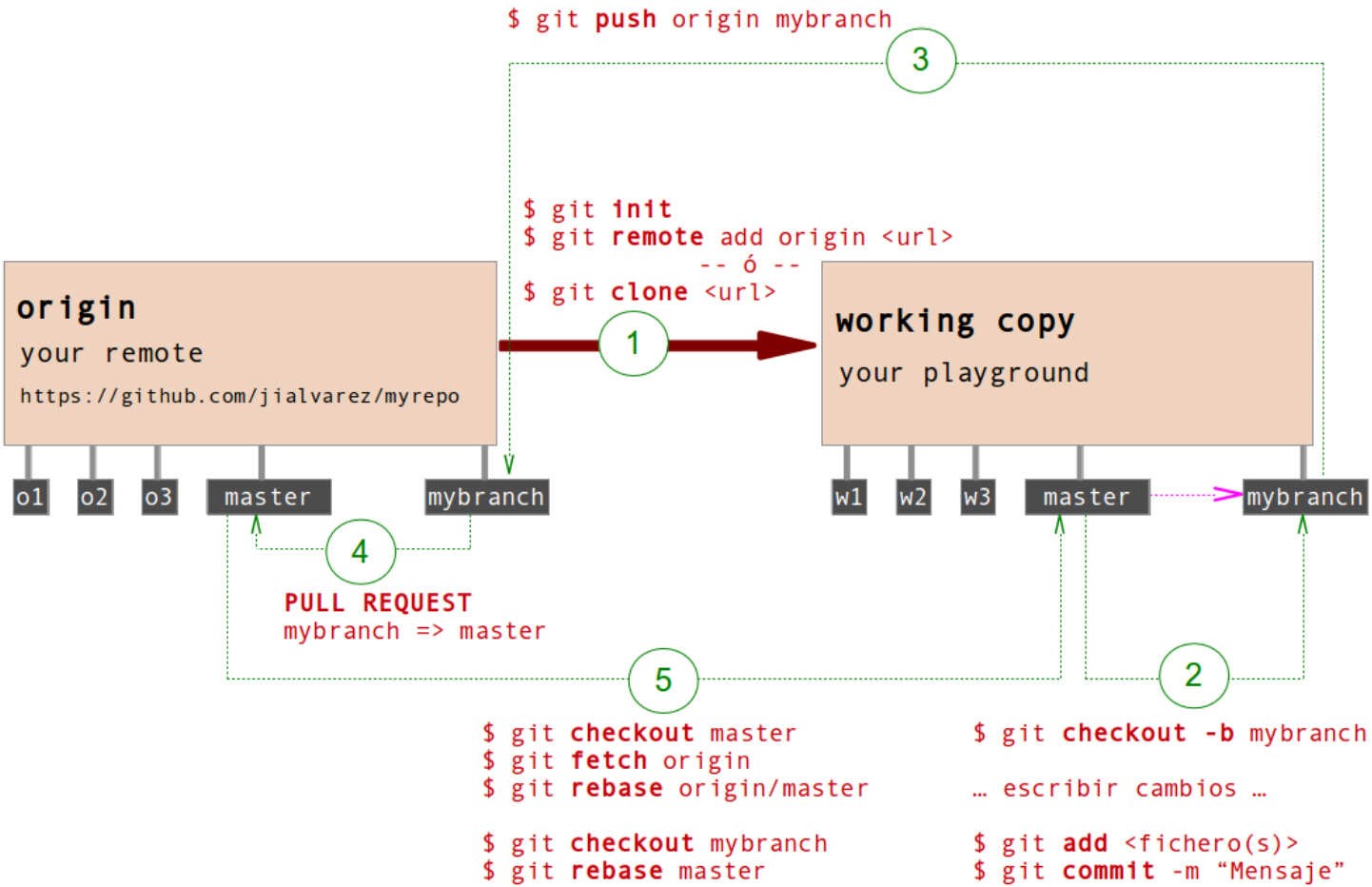


GIT

- Nacido de la mente de Linus Torvalds. Versión 1.0 en 2005.

- Se buscaba una manera de gestionar la ingente cantidad de código del kernel de Linux
- Toma el diseño de versiones anteriores de BitKeeper y Monotone
- Git se basa en *snapshots*, cada commit es una copia completa del código comprimida en binario, lo que le imprime velocidad. Usan compresión delta zlib para optimizar el espacio.
- Se ha medido git log frente a svn log, git opera 100x más rápido
- Se opera siempre en local (de ahí el incremento de velocidad) y cuando se tienen listos los cambios se suben a remoto

Flujo de trabajo en GIT



El puntero HEAD

- HEAD es una referencia simbólica que a menudo apunta al último commit de la rama actual
- A veces el HEAD apunta directamente a un objeto commit, en este caso el estado se llama *detached HEAD mode*. En este estado, incorporar un commit no reflejará cambios en rama alguna
- El primer precedesor de HEAD puede direccionarse vía HEAD~1, HEAD~2 y así sucesivamente. Si cambias entre ramas, el puntero HEAD se mueve al último commit de la rama de trabajo. Si se hace un *checkout* de un commit específico, el puntero HEAD apunta a este commit.

Etapas de un fichero (I)

1. Archivos **sin seguimiento**
2. Cambios **no preparados** para el commit
3. Cambios **para hacer** commit

```
# Archivos sin seguimiento:
# (use git add <archivo>... para incluir lo que se ha de ejecutar)
#
# fichero.tex
```

- Son archivos que aún no forman parte del repositorio local ni del remoto
- Al añadirlos pasan a la etapa III

Etapas de un fichero (II)

```
# Cambios no preparados para el commit:
# (use git add <archivo>... para actualizar lo que se ejecutará)
# (use git checkout -- <archivo>... para descartar cambios en el
directorio de trabajo)
#
# modificado: advanced.tex
# modificado: principal.pdf
```

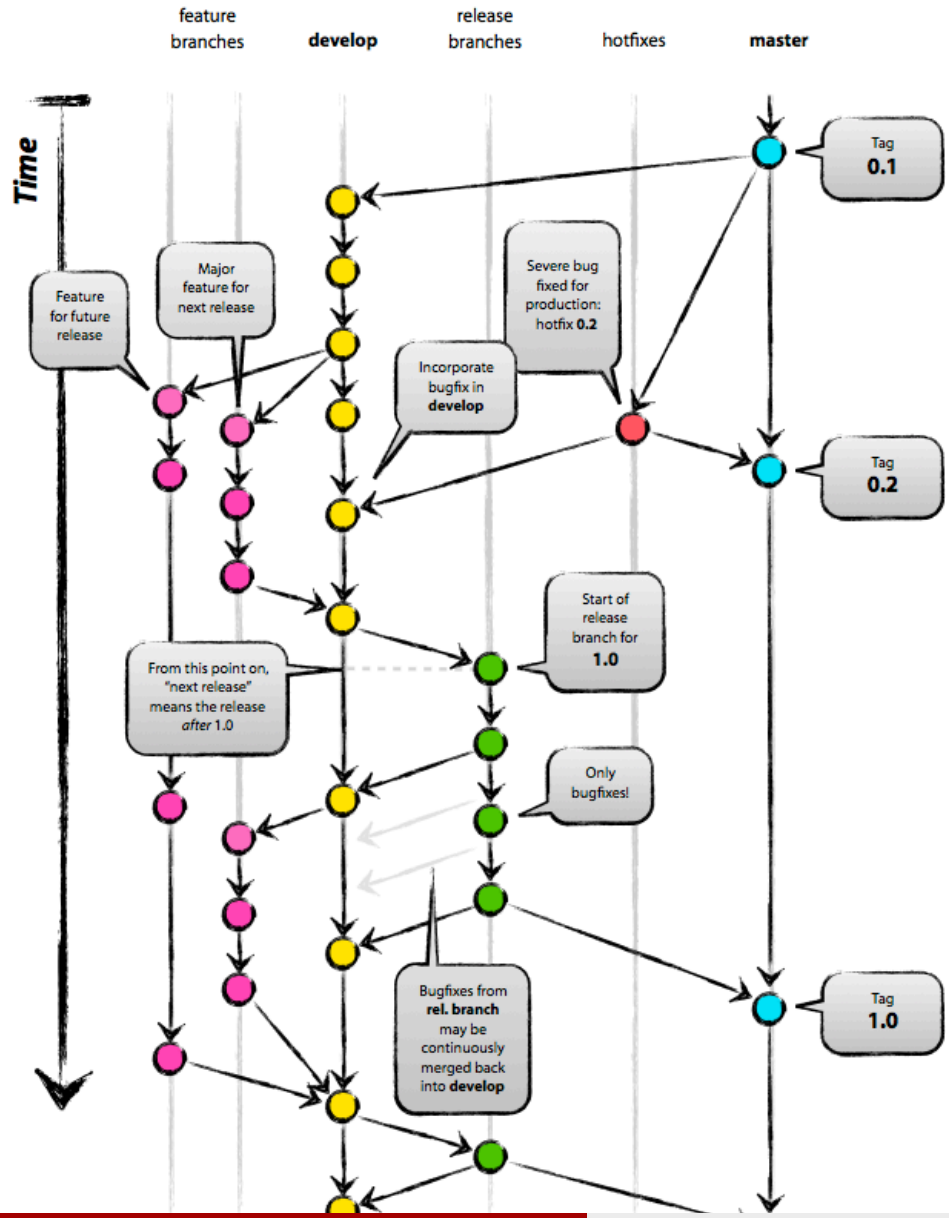
- Son ficheros ya añadidos previamente pero que aún no han sido *marcados* para comitear
- Al añadirlos pasan a la etapa III
- Si le aplicamos un *checkout* vuelven a la etapa I

Etapas de un fichero (III)

```
# Cambios para hacer commit:
# (use git reset HEAD <archivo>...para eliminar stage)
#
# modificado: workflow.tex
```

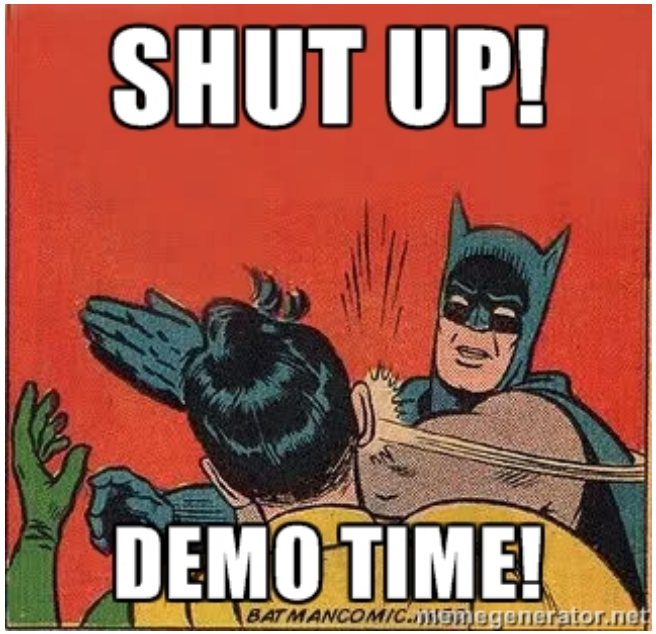
- Son ficheros ya añadidos previamente y marcados para comitear
- Si le aplicamos un *reset HEAD* vuelven a la etapa II
- En esta etapa es donde se puede ***comitear*** y hacer ***push***

Git branching



- Reglas propuestas por Vincent Driessen
- Ramas **master** (commits producción) y **develop** (siguiente versión planificada)
- Ramas **feature** (funcionalidad concreta). Se originan a partir de **develop** y vuelven a **develop**
- Ramas **release** (siguiente versión en producción). Se originan de **develop** y pasan a master o **develop**.
- Ramas **hotfix** (bugs en producción). Se originan a partir de **master** y vuelven a **master** o **develop**.

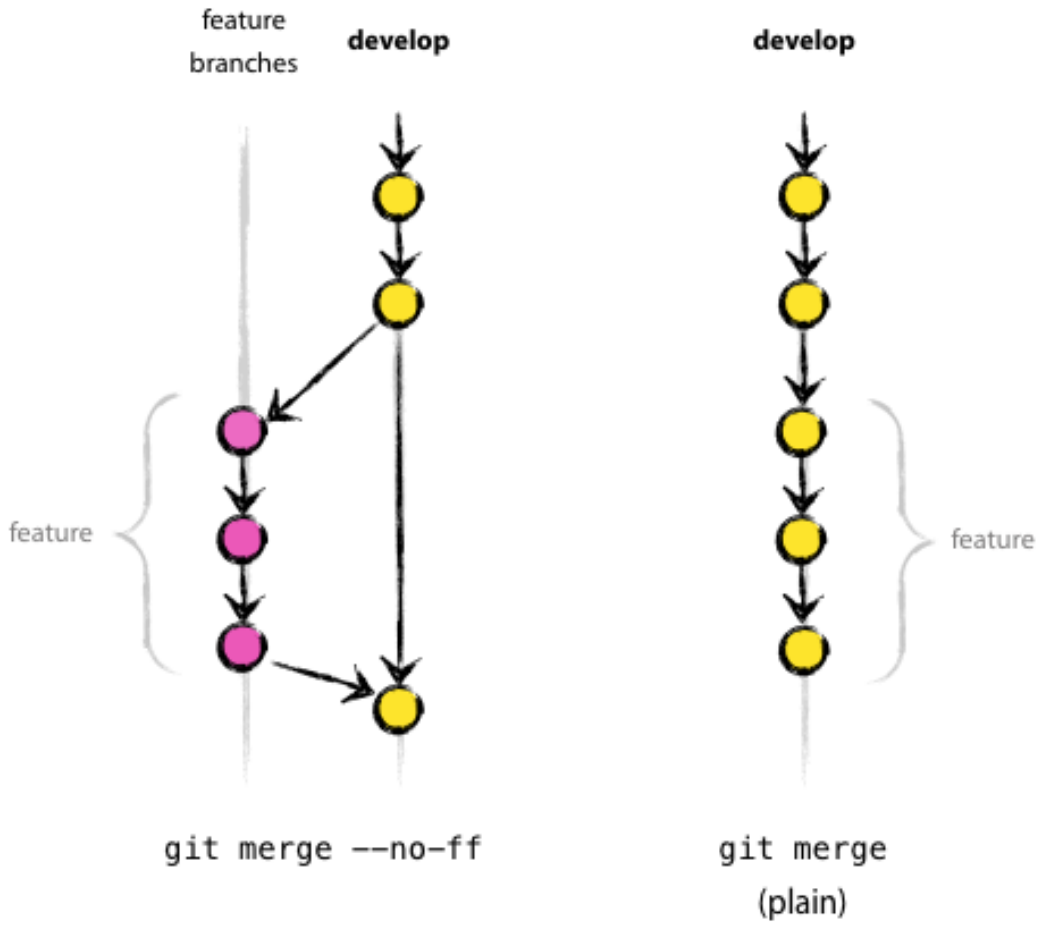
Demo



- Crearemos nuestro propio repositorio Git en Bitbucket
- Crearemos nuestra rama de trabajo y subiremos cambios
- Propondremos la integración de estos cambios haciendo *pull request*
- Simularemos la descarga del repositorio por parte de un compañero y subiremos nuevos cambios
- Actualizaremos el repo haciendo *rebase* y subiremos más cambios
- Mostraremos cómo funciona el *rebase* de manera gráfica

Merges en caso de conflicto

$ git **checkout** master
Switched to branch 'master'
$ git **merge** --no-ff -m 'Merged pull request #7' mybranch



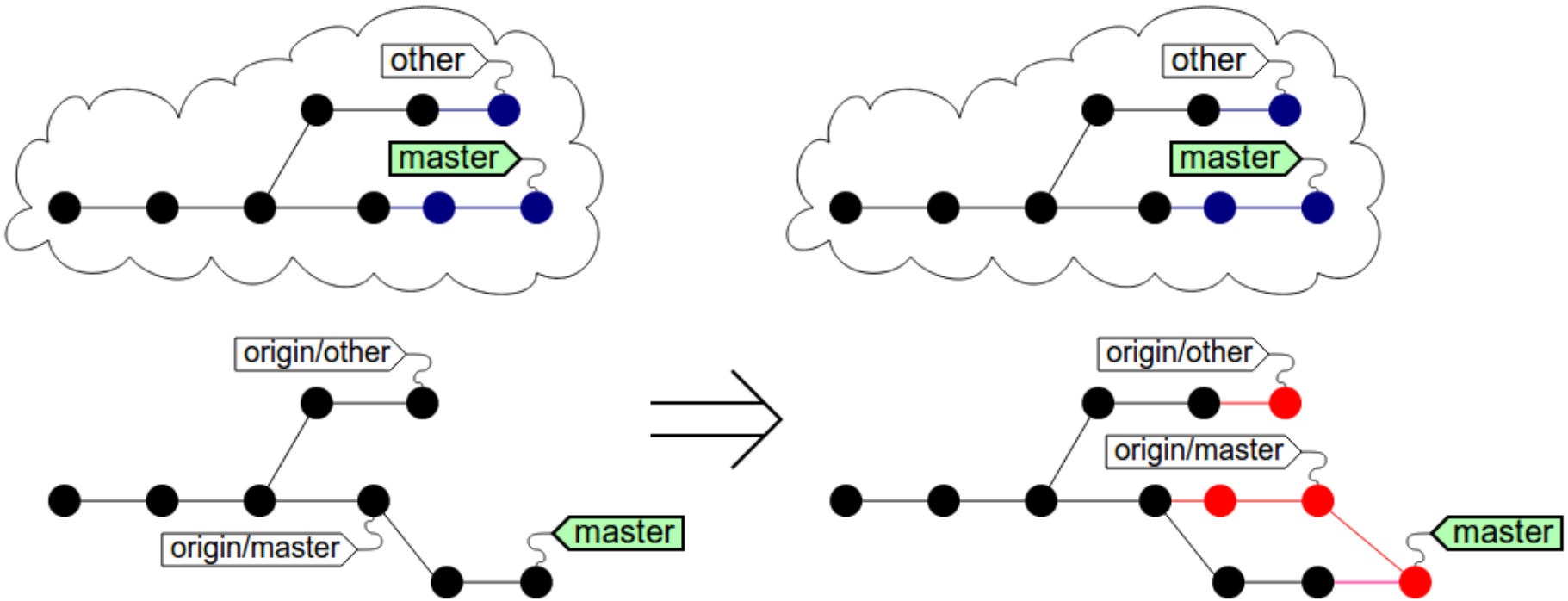
Tags

- En SVN, comúnmente se congela una versión copiando y pegando el *trunk* a la carpeta *tags* y subiendo esta nueva carpeta
- En Git, un tag no es más que una **etiqueta** en un commit concreto
- Con este simple acto, Git puede **recuperar una versión** en cualquier momento utilizando la etiqueta del tag
- Dos tipos:
  - **Lightweight tags:** git **tag** <etiqueta>
  - **Annotated tags:** git **tag** -a <etiqueta> -m <mensaje>
- Para recuperar un tag, basta con hacer git **checkout** <tag>

Más funcionalidades útiles

pull

Utilizándolo sin ningún parámetro adicional -**git pull origin master**- funciona igual que un *update* en Subversion. Es decir, se trae los cambios y hace *merge*. Si le pasamos el parámetro --*rebase*, funciona como un *rebase*.



ignore

El fichero *.gitignore* se suele colocar en la raíz del proyecto y el contenido suele ser un listado de elementos que no queremos que sean reconocidos como ficheros del repositorio.

```
neonigma@hyperion:~/things/taller-git$ cat .gitignore
*.aux
*.bbl
*.log
*.backup
*.toc
*.dvi
*.out
neonigma@hyperion:~/things/taller-git$ git log principal.aux
neonigma@hyperion:~/things/taller-git$
```

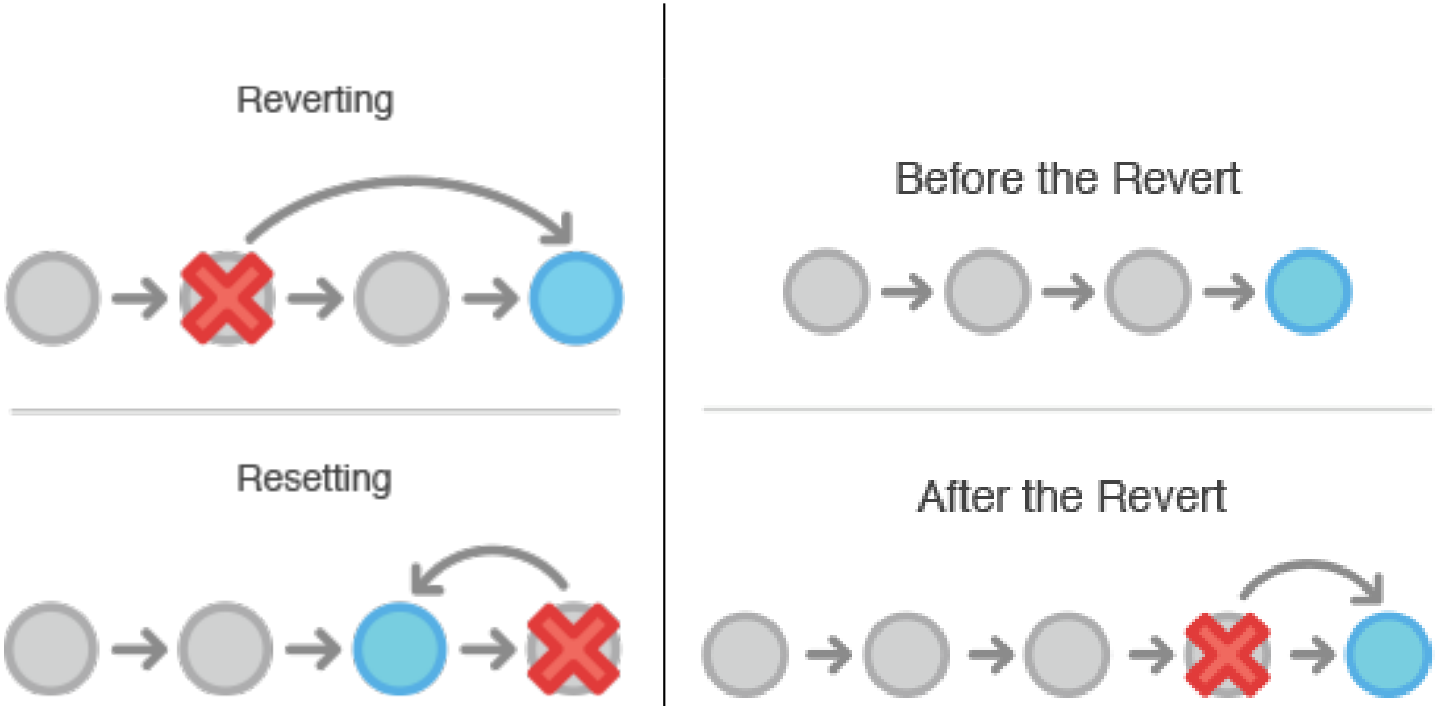
update-index

El comando **update-index** --assume-unchanged se utiliza para ficheros que accidentalmente se han *comiteado* (y probablemente *pusheado*) y no queremos tener en cuenta los cambios producidos en estos ficheros, ya que lo veremos como modificados en la etapa II (listo para *comitear*).

```
$ echo "Nueva linea\n" >> bibliografia.bib

$ git status
# En la rama master
# Cambios no preparados para el commit:
#   (use «git add <archivo>...» para actualizar lo que se ejecutará)
#  (use «git checkout -- <archivo>...« para descartar cambios en le directorio
#
# modificado:   bibliografia.bib
#

$ git update-index --assume-unchanged bibliografia.bib

$ git status
# En la rama master
nothing to commit, working directory clean
```

revert

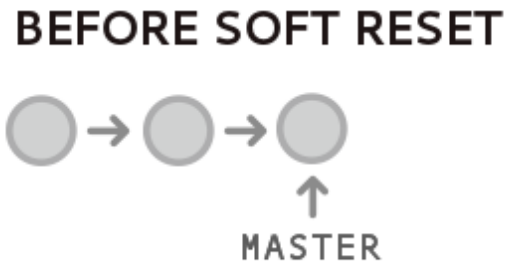
Este comando deshace un único commit aplicando el parche con la diferencia como un nuevo commit. Ejemplo: git **revert** HEAD

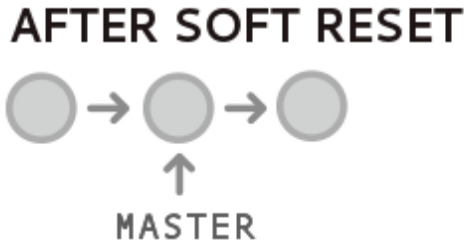


Este comando no destruye la historia ni diverge las ramas *master*. Ejemplo pack commits: git **revert** master~2..master

soft reset

El comando git **reset** --soft <hash> mueve el puntero de la cabeza al hash del commit que le indiquemos.

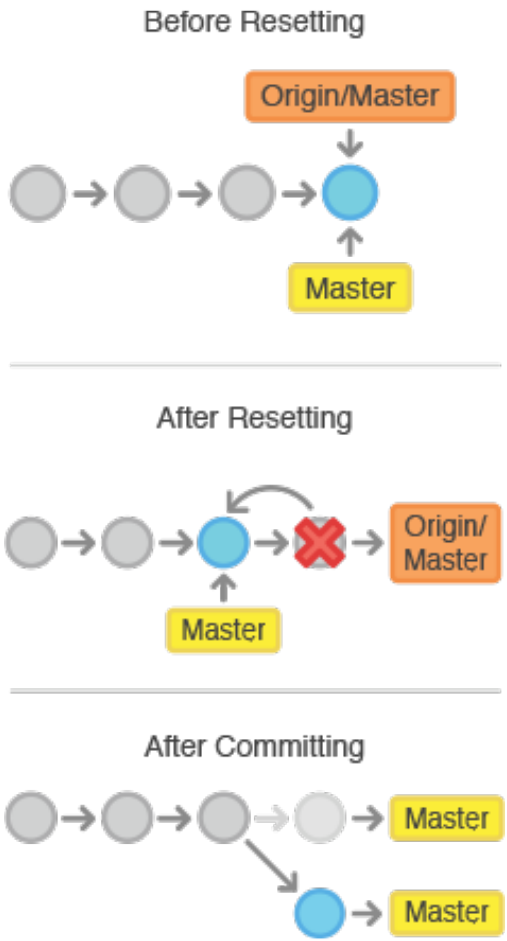




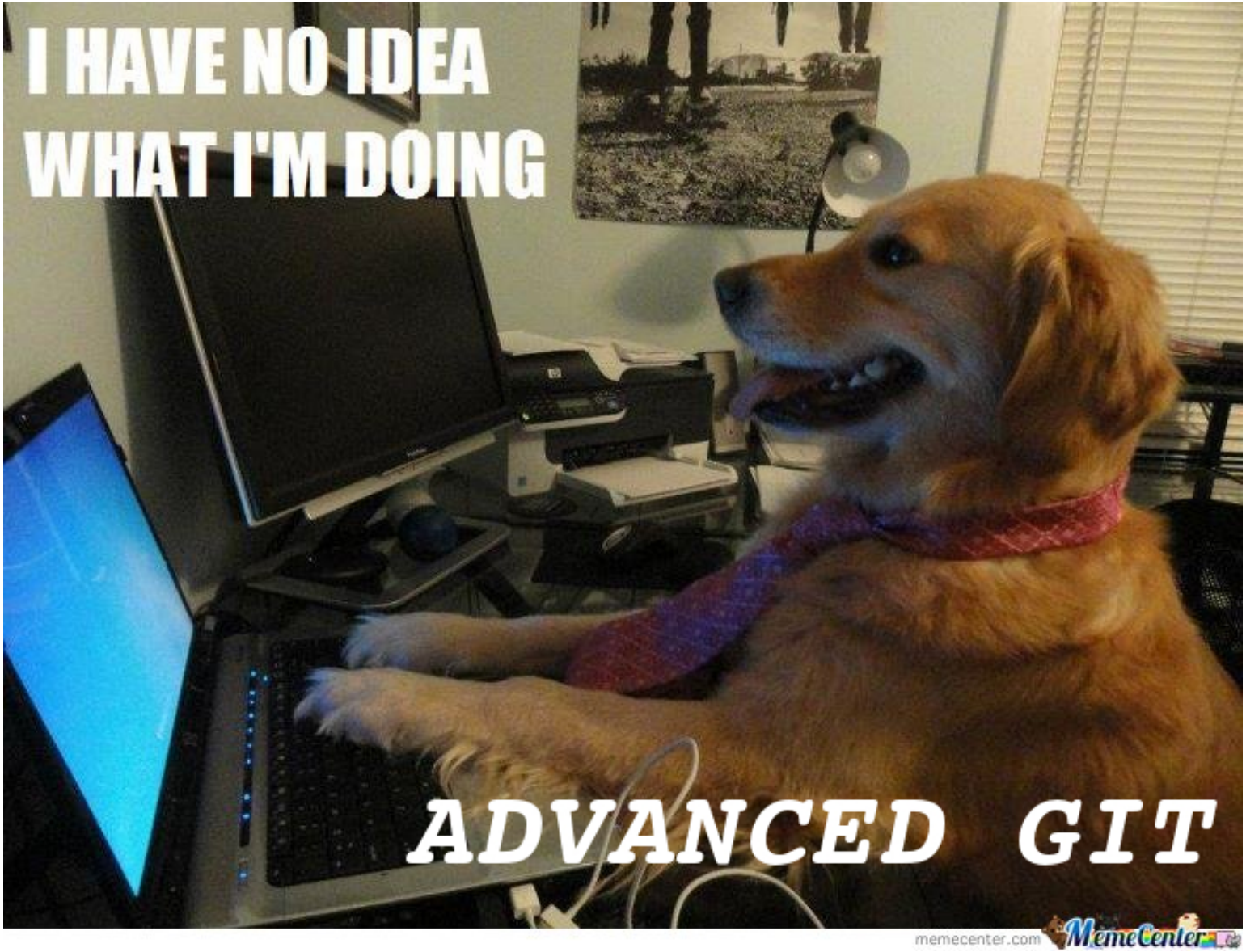
Esto provoca que los ficheros afectados entren de nueva en la etapa III (cambios listos para hacer commit).

hard reset

El comando git **reset** --hard <hash> mueve el puntero de la cabeza al hash del commit que le indiquemos y **destruye** toda la historia hasta dicho hash.



Algo más avanzado



stash

Se utiliza para *aparcar* temporalmente los cambios actuales antes de ser comiteados. El comando se escribe únicamente git **stash**.

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051... Revert added file_size
stash@{2}: WIP on master: 21d80a5... added number to log
```

Recuperamos los cambios con el comando git **stash** apply <id> o con git **stash** pop (tipo pila)

format-patch

Se utiliza para generar parches que se entregarán a mantenedores de repositorios que no aceptan *pull requests*.

Un parche se genera con el comando git **format-patch** --stdout <*hash o rango de hashes*>. Esto genera tantos ficheros con parches como commits se hayan introducido como parámetro.

Para aplicar los parches, se utiliza el comando git **am** --signoff < *file.patch*

squash

Se utiliza dentro de lo que se llama el *rebase interactivo*, para unir varios commits en uno solo antes de entregarlos al repositorio remoto. El proceso sería:

- Comiteamos tantas veces como queramos: git commit -m “mensaje”
- Si por ejemplo hemos hecho 2 commits, ejecutamos el rebase interactivo con: git **rebase** -i HEAD~2

```
pick 4ca2acc commit file1
pick 7b36971 commit file2
# rebase 41a72e6..7b36971 onto 41a72e6
#
# commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like “squash”, but discard this commit’s log message
# x, exec = run command (the rest of the line) using shell
```

cherry-pick

- Permite incorporar commits individuales a tu rama de trabajo
- El commit procederá de otra rama cualquiera
- La sintaxis: git **cherry-pick** <hash_commit>
- La ventaja de esto es aprovechar funcionalidades comiteadas por otros miembros del equipo
- Si la recuperación del commit trae problemas de mezclado, se puede abortar esta recuperación con git **cherry-pick** --abort

reflog

- Con git **log** vemos todos los commits realizados (historia)
- git **reflog** nos permite revisar todas las **acciones** realizadas
- Se puede utilizar para recuperar commits o ramas perdidas al utilizar git **reset**
- Los commits reseteados están disponibles durante 30 días, y los normales, durante 90 días

```
neonigma@hyperion: /things/taller-git$ git reflog
abec02f HEAD@0: merge foo: Merge made by recursive.
9bdbd83 HEAD@1: 9bdbd83: updating HEAD
2d90ece HEAD@2: merge foo: Fast-forward
9bdbd83 HEAD@3: checkout: moving from foo to master
2d90ece HEAD@4: commit: hello
9bdbd83 HEAD@5: checkout: moving from master to foo
```

submodules

- submodules permite *linkar* un repositorio dentro de otro
- Es una funcionalidad amada y odiada a partes iguales, por sus fuertes ventajas y desventajas
- Otras alternativas pasan por usar **subtree**, **gitslave** o **repo**
- Submodules es usado por:

submodules

- Añadir un repositorio git como submódulo
  - git **submodule** add [-b master] git@bitbucket.org:jialvarez/testproject.git
  - git **commit** -m "Añadimos submódulo testproject"
  - git **push** <remote> <branch>
- A partir de aquí el trabajo con el submódulo sería de manera normal, como un repositorio git cualquiera
- Tras actualizar el submódulo (add + commit + push), debemos actualizar también la referencia del padre de la misma manera
- ¡¡¡¡¡¡CUIDADO!!!!!!
  - Si hacemos *push* en el proyecto padre y no en el proyecto hijo, el siguiente que haga un *clone* del proyecto, no podrá descargarse todo el código al faltarle referencias
  - Si no pusheamos el hijo y hacemos un par de commits, y luego tratamos de actualizar el padre con git submodule update, dejamos al hijo en DETACHED mode (sin ramas)

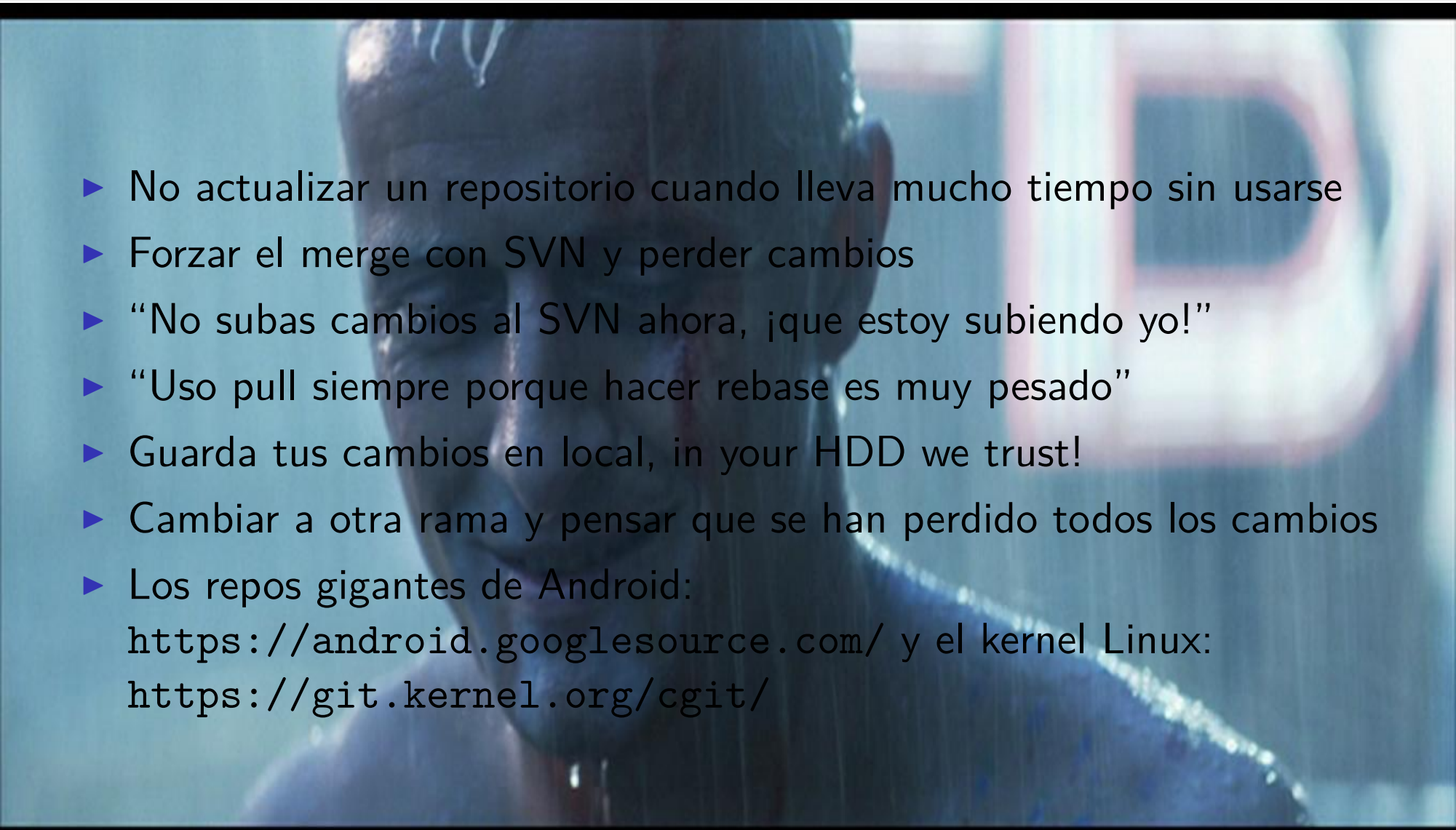
submodules

- Descargar / actualizar repositorio conteniendo submódulos
  - git **clone** git@bitbucket.org:jialvarez/testproject.git
  - git **submodule** init
  - git **submodule** update
- Para hacer un barrido por todos los módulos y actualizarlos:
  git **submodule** foreach git pull origin master
- Para borrar un submódulo:
  - Se eliminan sus líneas de configuración del fichero .gitmodules
  - Se eliminan sus líneas de configuración del fichero .git/config
  - Se ejecuta: git **rm** --cached <path/to/submodule>
- submodules es parte del *core* de git y está disponible para todos los SO y bien integrado en GUIs

lolcommits

- Te permite tomar una foto desde la Webcam de manera automática cada vez que haces git commit
- Puede descargarse aquí: `http://mroth.github.io/lolcommits/`
- Tiene una lista de plugins para cambiar el vocabulario de commits, subir automáticamente las imágenes a un servidor, tuitear, etc.
- Se pueden hacer gif animados con las imágenes tomadas: `http://nacho-alvarez.es/descargas/myimage.gif`
- En Linux y Mac OS X es ultrasencillo de instalar, en Windows algo más complejo
- Para habililitar lolcommits en un repositorio git, sólo hay que escribir **lolcommits** --enable
- Si queremos ver la última imagen: **lolcommits** --last , si queremos abrir la carpeta: **lolcommits** --browse

He visto cosas que no creeríais...



- No actualizar un repositorio cuando lleva mucho tiempo sin usarse
- Forzar el merge con SVN y perder cambios
- "No subas cambios al SVN ahora, ¡que estoy subiendo yo!"
- "Uso pull siempre porque hacer rebase es muy pesado"
- Guarda tus cambios en local, in your HDD we trust!
- Cambiar a otra rama y pensar que se han perdido todos los cambios
- Los repos gigantes de Android: `https://android.googlesource.com/` y el kernel Linux: `https://git.kernel.org/cgit/`

Herramientas

- **Intellij** (Windows, Mac, Linux)
  `http://www.jetbrains.com/idea/`
- **Eclipse** (Windows, Mac, Linux)
  `http://www.eclipse.org/`
- **SourceTree** (Windows, Mac)
  `http://www.sourcetreeapp.com/`
- **SmartGit** (Windows, Mac, Linux)
  `http://www.syntevo.com/smartgithg/`
- **Git Tower** (Mac)
  `http://www.git-tower.com/`
- **TortoiseGit** (Windows)
  `https://code.google.com/p/tortoisegit/`

Enlaces de interés

- **Git con calcetines**
  http://danielkummer.github.io/git-flow-cheatsheet/
  https://www.atlassian.com/git/tutorial/git-basics
  https://www.atlassian.com/git/workflows
- **Successful Git Branching model**
  http://nvie.com/posts/a-successful-git-branching-model/
- **Visual Git learning**
  http://pcottle.github.io/learnGitBranching/
- **Github vs Bitbucket**
  http://www.infoworld.com/d/application-development/
  bitbucket-vs-github-which-project-host-has-the-most-22706
- **Git Android**
  https://github.com/android
  https://android.googlesource.com/