Git

Escuela Tec. Sup. Ingeniería Telecomunicación

gsyc-profes (arroba) gsyc.es

Noviembre de 2015

Introducción a los sistemas de control de versiones

- En cualquier proyecto software de mediano tamaño trabajan muchas personas, en un número muy grande de ficheros, fuertemente dependientes entre sí
- Los ficheros cambian continuamente, cada fichero irá teniendo distintas *revisiones*: r23, r24, r25...
- Si solo se cuenta con un directorio compartido, los problemas son enormes
  Ejemplo 1:
  - Juan modifica `client.c r24` y genera `client.c r25`, que interactúa con `server.c r51`
  - Pero María edita `server.c r51` que pasa a ser `server.c r52`, introduciendo cambios que provocan que el cliente deje de funcionar

Ejemplo 2:

- Despues de cientos de revisiones de cada fichero, liberamos (publicamos) la primera versión del sistema disponible para nuestros usuarios, `miproducto v1.0`
- Liberamos una nueva versión `v1.1`, que incluye funcionalidad adicional, no muy bien probada
- Aparece un problema en `miproducto v1.0`, por lo que preparamos un parche y liberamos la versión `v1.0.1`
- Es necesario aplicar un parche similar a `v1.1`, y con cambios adicionales, liberamos `v1.2`, que ya consideramos estable
- Vamos trabajando en una futura versión `v2.0`, que supondrá un rediseño completo. Pero simultánemente, seguiremos desarrollando la versión `1.3`

Resulta evidente la necesidad de un *sistema de control de versiones*, también llamado *repositorio* que permita:

- Que cada desarrollador tenga su propia copia local *congelada* del sistema, pero que luego pueda llevar sus cambios al sistema global (hacer un *commit*)
- Conocer todos y cada uno de los cambios realizados: autor, fecha, diferencias con el estado anterior, etc
- Almacenar todas las revisiones y versiones, para poder recuperar o deshacer cualquiera de ellas
- etc etc

El uso más habitual es el desarrollo de software, pero es útil para cualquier entorno donde deseemos controlar los sucesivos estados un sistema de ficheros: ficheros de configuración, documentación diversa o incluso nuestro *home*

Evolución de los sistemas de control de versiones

VCS: *Version Control System*, aka RCS, *Revision Control System*

- Primera generación. RCS (Año 1982). No era un verdadero control de versiones sino de *revisiones*. El sistema podía gestionar todas las revisiones de `server.c` y de `client.c`, pero no sabía qué revisión de `client.c` era contemporánea a cada revisión de `server.c`
- Segunda generación. CVS (Año 1990). Resuelve el problema anterior, entre otros

- Tercera generación. SVN (Año 2000). Diversas mejoras: atomicidad, mantenimiento del historial tras los cambios (renombrar, copiar, mover), mayor eficiencia con los binarios, creación de ramas, soporte unicode...
- Cuarta generación. Sistemas distribuidos de control de versiones (DVCS, *Distributed Version Control System*). Git (Linux Torvalds, año 2005). Mercurial (Año 2005). En principio Git era más potente y Mercurial más sencillo de manejar, pero ambos han mejorado mucho y actualmente resultan similares

Ventajas de git frente a sistemas de generaciones anteriores

- Por ser distribuido, toda la gestión se hace en el repositorio local. Luego, se puede integrar mediante *push* y *pull* con uno o varios repositorios remotos
- Un repositorio es un único directorio, autocontenido, con todo el historial completo.
  Eso lo hace muy robusto frente a fallos: aunque es habitual emplear una topología en estrella con un maestro y varios esclavos, cualquiera de los esclavos puede reemplazar al maestro
- Cada estado se almacena completo. No como la diferencia desde el estado anterior

Otras características de git

- Los commit del repositorio local se pueden modificar y rehacer por completo, de manera que los commit que subimos al repositorio maestro estén muy cuidados
  Pero incluso se pueden subir cosas que modifiquen lo ya subido. O puede desaparecer parte del historial, creando *commit fantasma*
  Los defensores de mercurial critican esta capacidad de *modificar el pasado*

Instalación de git

Para debian y derivados:

1. `apt-get update  # Refresca la lista de paquetes`
2. `apt-get upgrade # Actualiza el sistema`
3. `apt-get install git-core`

La instalación es el paso 3, los pasos 1 y 2 normalmente son recomendables antes de instalar cualquier paquete
Naturalmente, esto solo lo puede hacer el administrador. Y en el laboratorio no será necesario, git ya está instalado
Clientes gráficos:

- Aquí veremos como se usa git desde la shell, pero también podemos instalar clientes gráficos
  En Windows y MacOS podemos emplear p.e. `sourcetree` (*freeware*)

Configuración inicial

- Antes de empezar a usar git, es necesario indicar nuestro nombre y correo electrónico, para que conste en cada commit
- Es conveniente añadir algunos *alias*, para simplificar el uso de las órdenes más habituales
- Para todo ello, recomendamos crear un fichero `~/.gitconfig` con el siguiente contenido, modificando los datos personales:

```
[alias]
    log2 = log --pretty=format:\"%h %ad | %s%d [%an]\" --graph --date=short
    log3 = log  --pretty=format:\"%h %ad | %s%d [%an]\"  --name-status  --date=short
    log4= log --color --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s \
          %Cgreen(%cr)  %C(bold blue)<%an>%Creset' --abbrev-commit
    ca = commit --all
    caa = commit  --all --amend
    co = checkout
    br = branch
[user]
    name = Juan García García
    email = juan.garcia@micorreo.com
[push]
    default = matching
```

Puedes descargar este fichero de configuración en `http://tinyurl.com/nancr6j`

- Observa que aquí usaremos órdenes como
  `git co`, `git ca`, `git br`, `git log2`,
  que no son estándar, sino alias personalizados que no funcionarán a menos que los definas en este fichero

Creación de un repositorio

Un repositorio es un directorio: normalmente el VCS controla todos los ficheros (y directorios) de su interior, excepto aquellos que no incluyamos
Primero trabajaremos con un repositorio local, posteriormente lo integraremos con repositorios remotos
Hay varias formas de crear un repositorio, veremos dos

- Creación partiendo de cero
- Creación mediante clonación por ssh
  Consiste en crear un repositorio en la máquina local que es un clon de un repositorio que está en una máquina remota a la que tenemos acceso por ssh

Otras posibilidades son la clonación de un directorio local o la clonación de un directorio remoto a través de http

Creación partiendo de cero

Creamos un directorio vacío, entramos dentro e invocamos a `init`

```
mkdir repo01
cd repo01
git init
```

Creación mediante clonación por ssh

Creamos un directorio vacío y desde el directorio padre del directorio recién creado, invocamos a `git clone`, pasando como primer argumento la dirección del repositorio preexistente y como segundo argumento, el directorio (vacío) que acabamos de crear

```
mkdir repo01
git clone usuario@maquina:/var/tmp/padre.git  repo01
```

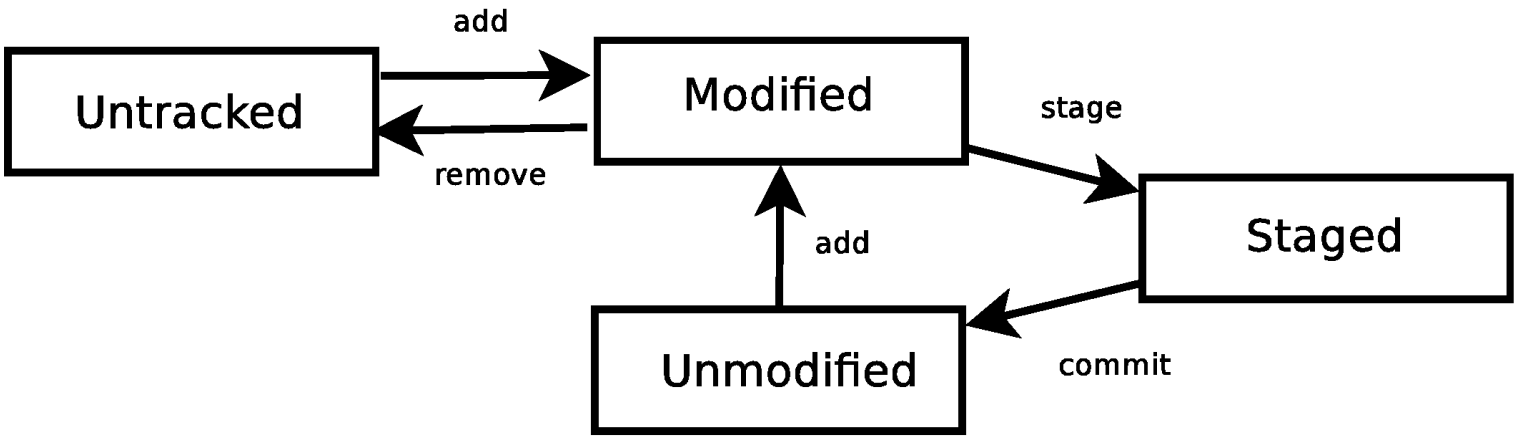
Ficheros de configuración

- Todo repositorio git tiene un subdirectorio oculto `.git` que contiene su historial completo. Si copiamos/movemos el directorio del repositorio junto con el subdirectorio `.git`, estamos copiando/moviendo el repositorio completo, con todas sus características
- El fichero `.git/config` contiene la configuración de ese repositorio
  Por tanto, git maneja dos ficheros de configuración
  1. `~/.gitconfig`
     Fichero de configuración global, aplicable a todos los repositorios del usuario en esa máquina
  2. `.git/config`
     Fichero de configuración particular de cada repositorio
     p.e. `~/repo01/.git/config`

- Podemos incluir dentro del repositorio un fichero `.gitignore`, que especifica qué ficheros se deben ignorar p.e.

```
*.log
*.out
*.txt~
.*
```

Estados de un fichero en git



- *untracked*: fichero que el repositorio ignora
- *unmodified*: fichero que está en el repositorio
- *modified*: fichero del repositorio que ha cambiado, aunque el cambio de momento se ignora
- *staged*: fichero del repositorio que ha cambiado, y que será actualizado en el repositorio en el próximo *commit*

Marcar ficheros para su seguimiento

Un fichero recién creado o copiado en el directorio, inicialmente es ignorado por git, su estado es *untracked*
El primer paso será marcarlo para que git lo siga, lo ponemos en estado "modified"

```
echo "primera linea" > a.txt
git add a.txt  # Pasa el fichero a "modified"
```

Inclusión de ficheros en el repositorio

Git introduce el estado *modified*, que no tiene equivalente en VCS anteriores

Supongamos que trabajo en los ficheros `a.py`, `b.py`, y `c.py`

1. `a.py` y `b.py` están listos para el commit.
2. `c.py` todavía no
3. Quiero hacer ya un commit, sin esperar a terminar `c.py`

¿Cuál es el estado de `c.py`?

- No puede ser *unmodified*, porque ya no está como estaba en el commit anterior
- No puede ser *staged* porque no quiero incluirlo en el próximo commit
- Tampoco quiero pasarlo a *untracked*, porque eso haría que el repositorio dejase de seguir el fichero (incluyendo la versión que ya está en el repositorio)

El estado de `c.py` es *modified*

- Como hemos visto, el estado *modified* puede ser conveniente en ciertos escenarios
  1. Los ficheros modificados automáticamente pasan a *modified*
  2. Los que queramos llevar al commit, los pasamos a *staged* con la orden `git add`
  3. Hacemos commit

```
git add a.py b.py  # Lleva los ficheros a "staged"
git commit         # Lleva los ficheros a "unmodified"
```

Para deshacer esto y que un fichero vuelva a *modified*

```
git rm --cached b.py
```

- Pero también se puede *vivir sin el estado modified*, como hacían todos los VCS anteriores

```
git commit --all   # Lleva todos los ficheros a "unmodified"
                   # (excepto los "untracked")
```

O con el alias que hemos definido en `~/gitconfig`

```
git ca
```

- Por simplificar, en esta asignatura trabajaremos siempre de la segunda forma,
  `git ca`
  ignorando la diferencia entre *modified* y *staged*
- Esto resulta más cómodo porque no es necesario hacer un
  `git add`
  para cada fichero en cada commit
  Basta con hacer `git add` una sola vez al crear el fichero, para sacarlo del estado *untracked*

Es impresincible que todo commit tenga un comentario
Se puede indicar de dos maneras

- Con la opción `-m`
  `git ca -m "Modifica tal y cual cosa"`
- Sin indicarlo desde la shell
  `git ca`
  En este caso, git abrirá nuestro editor por omisión para que introduzcamos el comentario
  - El editor por omisión podemos especificarlo definiendo la variebla de entorno `EDITOR` en el fichero `~/.bashrc`

    ```
    export EDITOR=vim
    ```

La recomendación es escribir los comentarios en presente y en tercera persona, como si todos empezaran por *Este commit...*

Ejemplo

```
mkdir mi_repo
cd mi_repo
git init  # Creamos el repositorio partiendo de cero
echo "Primera linea" > a.txt
git add a.txt    # Pasa el fichero a "modified"
git ca -m "Crea el primer commit de prueba"
echo "Segunda linea" >> a.txt
git ca -m "Crea el segundo commit de prueba"
```

Con esto hemos creado un repositorio, le hemos añadido un fichero, hemos hecho un commit, lo hemos modificado y hemos realizado un segundo commit

La orden

`git status`

devuelve el estado de todos los ficheros del directorio actual y sus subdirectorios

```
koji@mazinger:/tmp/mi_repo$ git status
# En la rama master
nothing to commit, working directory clean
koji@mazinger:/tmp/mi_repo$ echo blabla > b.txt
koji@mazinger:/tmp/mi_repo$ echo "segunda linea" >> a.txt
koji@mazinger:/tmp/mi_repo$ 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:   a.txt
#
# Archivos sin seguimiento:
#   (use «git add <archivo>...» para incluir lo que se ha de ejecutar)
#
# b.txt
no hay cambios agregados al commit (use «git add» o «git commit -a»)
```

La orden

`git log`

devuelve el registro de todos los commit del repositorio

```
koji@mazinger:/tmp/mi_repo$ git log
commit 6d6de6bb15bffae7bdd222b661905b14d342db4e
Author: Juan Pérez <jperez@urjc.es>
Date:   Mon Oct 28 18:07:15 2013 +0100

    Crea el segundo commit de prueba

commit 1681c2a4867c759eb1300d2b6514b7996071ab4c
Author: Juan Pérez <jperez@urjc.es>
Date:   Mon Oct 28 18:07:14 2013 +0100

    Crea el primer commit de prueba
```

En el fichero `~/.gitconfig` hemos creado los alias `log2`, `log3` y `log4`, que presentan diversas variantes de *logs* más compactos

```
koji@mazinger:/tmp/mi_repo$ git log2
* 6d6de6b 2013-10-28 | Crea el segundo commit (HEAD, master) [Juan Pérez]
* 1681c2a 2013-10-28 | Crea el primer commit de prueba [Juan Pérez]
```

También podemos usar la aplicación `gitk`, que permite navegar gráficamente por el repositorio

- `git rm nombre_fichero`
  Borra el fichero del repositorio
- `git mv nombre_viejo nombre_nuevo`
  Cambia el nombre del fichero

Objetos commit y ramas

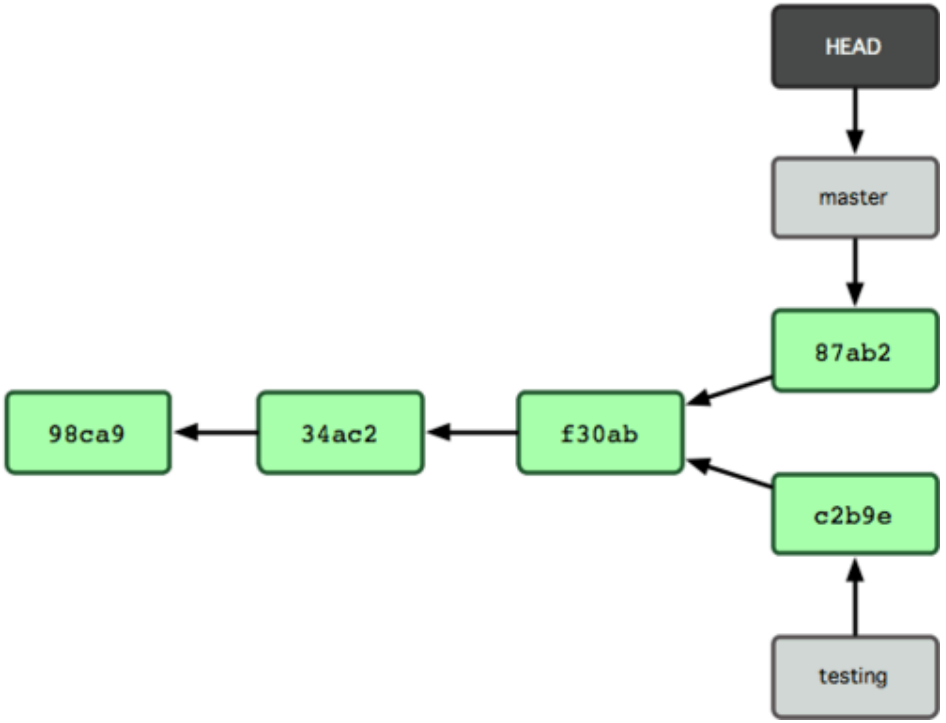


Gráfico:git-scm.com

- Cada caja verde respresenta un estado de sistema de ficheros. Git le llama *objeto commit*, informalmente podemos llamarle *foto* (*snapshot*)
- Cada objeto commit apunta al objeto del que procede, su objeto *padre*. Por tanto, el tiempo avanza en sentido contrario a las flechas

- Cada objeto commit se identifica por un *checksum* SHA1
- Si cada *padre* tiene un solo *hijo*, el desarrollo es lineal y el sistema tiene una única rama (*branch*): la rama *master*
- En cierto momento del desarrollo, un padre puede tener un segundo hijo. Esto crea una nueva rama, independiente de la anterior, con desarrollo paralelo. En el ejemplo anterior, la rama se llama *testing*
- Para saber en qué rama estamos en cada momento, git emplea un puntero, llamado *HEAD*
  No debemos confundir HEAD con *master*
  - *master*: primera rama, rama principal, rama por omisión
  - HEAD: rama en la que estamos actualmente

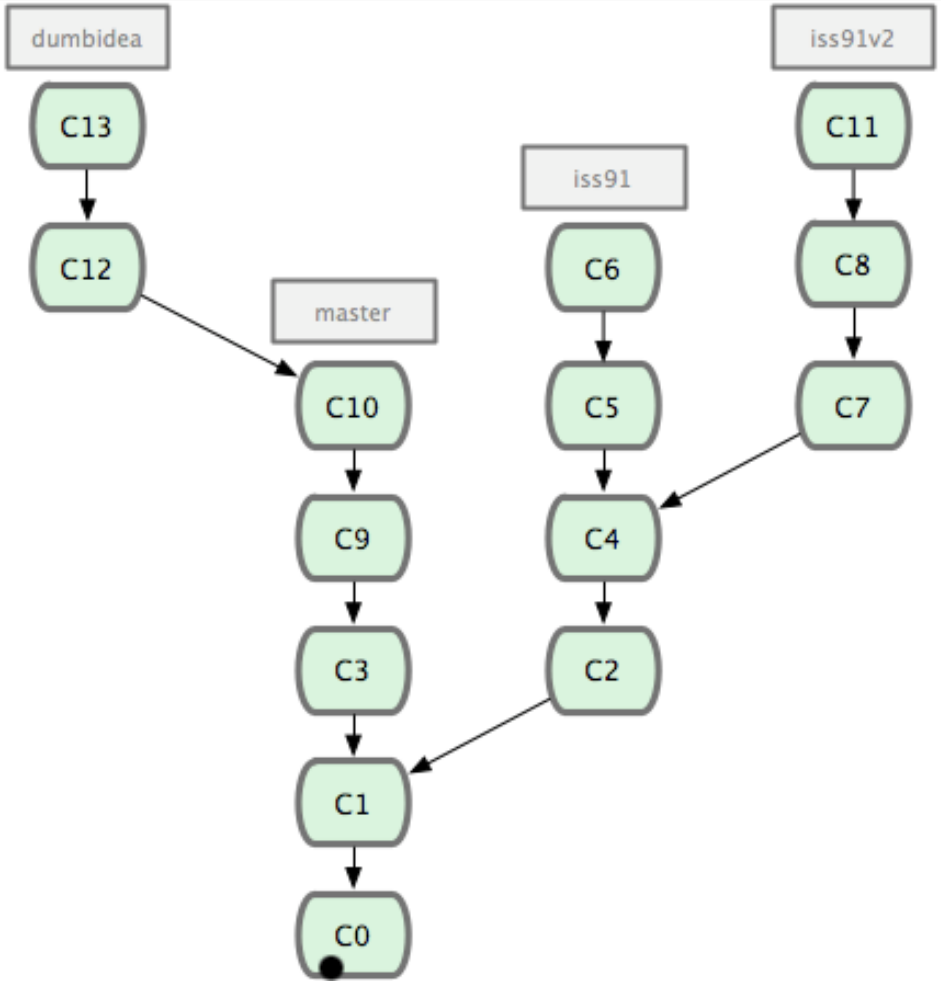


Gráfico:git-scm.com

- En otros sistemas de control de versiones, crear una rama es bastante costoso
- Pero en git las ramas son muy eficientes y baratas, git anima a crear muchas ramas, que luego pueden fusionarse automáticamente

- `git br` [1]
  Muestra todas las ramas. Marca con un asterisco la actual
- `git br nueva_rama`
  Crea una rama nueva (pero no pasa a ella, HEAD se mantiene donde estaba)
- `git co nueva_rama`
  Pasa a la nueva rama (hace que HEAD apunte a la rama)

[1] Recuerda que br y co no son órdenes estándar de git, sino alias que hemos creado para branch y checkout

Recuperación de commits antiguos

Al recuperar un estado antiguo, el sistema queda en una situación especial. Obviamente, un estado pasado no se puede modificar sin más, porque los hijos de ese estado resultarían inconsistentes

- Una solución (que no usa git) podría ser dejar los ficheros en modo *read only*
- Lo que hace git es distinto: al retroceder a un estado pasado, el sistema sale de la rama en la que se encontraba y pasa a un estado especial, no asociado a ninguna rama

```
koji@mazinger:/tmp/repo.01$ git co a3f9ba6c6f6e
Note: checking out 'a3f9ba6c6f6e'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at a3f9ba6... Modifica esto y lo otro
```

- En un *viaje al pasado* podemos consultar o modificar cualquier cosa, pero los cambios no se almacenan en ningun sitio. A menos que decidamos crear una nueva rama (y pasar a ella)
  `git br rama_nueva`
  `git co rama_nueva` [2]
- Para *volver al presente*, extraemos la rama que deseemos. Por ejemplo, la rama master
  `git co master`

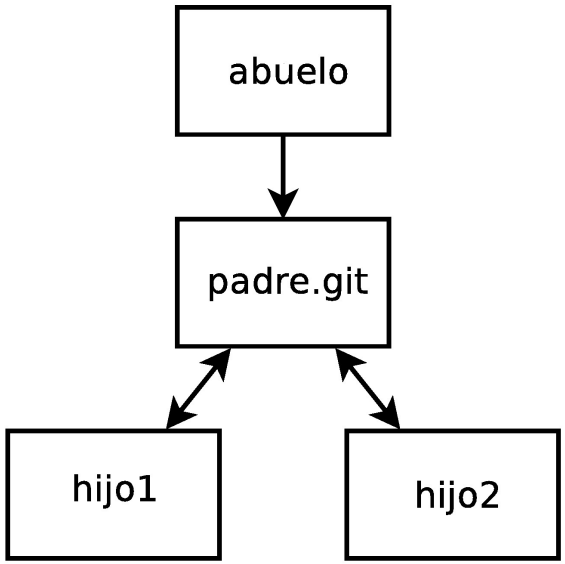
[2] Recuerda que br y co no son órdenes estándar de git, sino alias que hemos creado para branch y checkout

Repositorios remotos

- La principal característica de los VCS de 4ª generación es que son distribuidos: un repositorio local se puede sincronizar con un repositorio remoto
- Los repositorios se pueden organizar con muy diversas topologías, pero la básica es una topología en estrella con 1 maestro y n esclavos
  - Topologías más complejas suelen incluir varios niveles jerárquicos de maestros
  - Los esclavos no son simples réplicas del maestro, los cambios de los esclavos se propagan al maestro
    El maestro es maestro porque todos los cambios pasan a través de él

Un repositorio *maestro* es un repositorio un poco especial porque es de tipo *bare* (vacío)

- Se trata de un repositorio contra el que no se pueden hacer ni recuperar commits directamente, solo se usa para sincronizar otros repositorios
- No tiene ficheros ordinarios a la vista, solo contiene los ficheros que normalmente están dentro del directorio .git
- Por convenio, los repositorios maestros son directorios con la extensión `.git`



- Al repositorio maestro le llamaremos *padre.git*
  Es un repositorio especial, de tipo *bare*
- A los repositorios esclavos les llamaremos *hijos*
- El repositorio *padre.git* se crea a partir de otro repositorio al que llamaremos *abuelo*
  Una vez que el abuelo ha creado (por clonación) al padre, el abuelo se puede borrar
  - O se puede cambiar la configuración del abuelo para que *olvide* lo que era y pase a ser un hijo más

Si queremos que un hijo se sincronice contra otro hijo

- No podemos hacerlo directamente, siempre tiene que hacerse a través de un `padre.git`, de tipo *bare*
- Podemos hacer que este hijo tenga *descendencia* y pase a ser un *abuelo2*, de forma que todo el proceso se repita

Creación de un repositorio *abuelo* y un repositorio *padre*

```
#Creamos los directorios vacíos
mkdir abuelo
mkdir padre.git

# Entramos en el abuelo y hacemos de él un git
cd abuelo
git init

# En el abuelo creamos al menos un commit
# si no, el nieto tendrá problemas
touch primer_fichero
git add primer_fichero
git ca -m "primer commit"

# Salimos del abuelo y creamos el padre
cd ..
git clone --bare abuelo padre.git
```

1. Creamos el repositorio hijo clonando el padre
   - El hijo puede tener acceso al padre o bien porque esté en el mismo sistema de ficheros

     ```
     mkdir hijo
     git clone padre.git hijo
     ```

   - O bien porque tiene acceso por ssh

     ```
     mkdir hijo
     git clone usuario@maquina:/var/tmp/padre.git   hijo
     ```

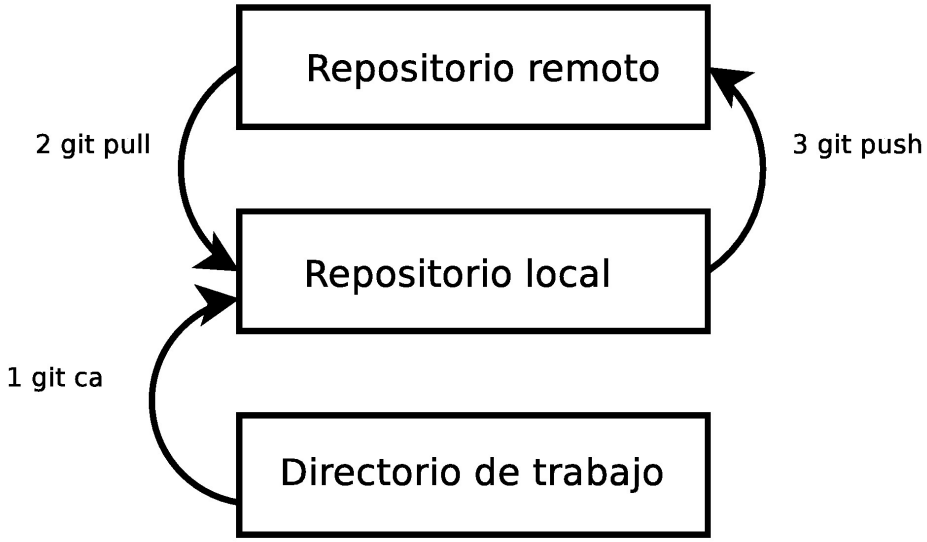
   - O bien a través de HTTP, HTTPS o mediante un protocolo propio de git para acceso remoto
2. Antes de usar el hijo por primera vez, es necesario indicar qué rama del padre usará[3]

   ```
   cd hijo
   git pull origin master
   git push origin master
   ```

[3] Si en el padre no hay ningún commit, este paso dará un error

Una vez que el repositorio hijo está enlazado con el padre, podemos ejecutar (dentro del hijo)

- `git pull`
  Baja los cambios (*tira*)
- `git push`
  Sube los cambios (*empuja*)



Cada vez que queramos sincronizar el hijo con el padre, es recomendable seguir estos tres pasos: [4]

```
git ca    # Hago commit de todos los cambios
git pull  # Bajo (e integro) los cambios del padre
git push  # subo los cambios al padre
```

[4] No serán todos necesarios si en el repositorio remoto no ha habido cambios o si hemos modificado ficheros no conflictivos, pero es conveniente adquirir este hábito, al menos inicialmente

Si olvidamos hacer `ca` antes de `pull`, es probable que obtengamos el siguiente error

```
error: Your local changes to the following files would be overwritten by merge:
    fichero
Please, commit your changes or stash them before you can merge.
Aborting
```

Git se niega a bajar el fichero, porque eso machacaría un cambio local que no ha entrado en el repositorio

Si olvidamos el `pull` antes de el `push`, es similar a decirle al repositorio padre *aquí va mi repositorio, intégralo con el tuyo*
Esto no es válido, probablemente obtendremos el siguiente error

```
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to '/var/tmp/padre.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details
```

Como hemos visto, lo correcto es hacer
`git ca; git pull; git push`
que equivale a decir *meto mis ficheros en mi repositorio, dame tus cambios, yo los integro, cuando lo tenga ordenado, lo subo*

Si queremos que el hijo se sincronice contra un padre en una ubicación distinta, basta editar la línea `url` en el fichero de configuración local del repositorio (`.git/config`)

```
[remote "origin"]
    fetch = +refs/heads/*:refs/remotes/origin/*
    url = ssh://milogin@192.168.1.12/var/tmp/padre.git
[branch "master"]
    remote = origin
    merge = refs/heads/master
```

- Si queremos que el abuelo pase a ser un hijo más, basta editar su fichero `.git/config` para añadir los dos párrafos anteriores

Conflictos

Si dos hijos editan en paralelo el mismo fichero, se provocará un conflicto que solo el usuario, a mano, podrá resolver

```
auto-merging hola
CONFLICTO(contenido): conflicto de fusión en hola
Automatic merge failed; fix conflicts and then commit the result.
```

Si se trata de ficheros de texto, git combinará ambos. Los bloques discrepantes estarán enmarcados entre los símbolos <<<<<<<, ======= y >>>>>>>
El usuario debe editar el fichero, añadirlo al repositorio y sincronizarlos

```
hola
<<<<<<< HEAD
Esta línea la escribo en hijo2
=======
Esta línea la escribo en hijo1
>>>>>>> 305e0f3a798338a9c990ffda032fe6322150102d
```

Referencias

- Scott Chacon. Pro Git book.
  Disponible en `http://git-scm.com/book`
- Git Immersion
  `http://gitimmersion.com`