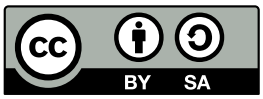
Colaboración con Git

Antonio García Domínguez
antonio.garciadominguez@uca.es

24 de septiembre de 2014



Índice

Índice de figuras

1. Acciones básicas

1.1. Clonado de repositorios

Cuando varias personas han de colaborar en un mismo proyecto en el que se use Git, normalmente usarán un clon de algún repositorio destacado como punto inicial, creado mediante una orden del estilo de `git clone url`, donde la URL variará según qué protocolo usemos. Una vez se cree el clon, podemos indicar a Git la URL desde que clonamos utilizando el identificador «origin». Éste es un ejemplo de un «remote» de Git: una referencia por nombre a una URL con ciertos atributos adicionales configurables. Podemos añadir los nuestros propios mediante las órdenes `git remote add nombre url` y retirarlos con `git remote rm nombre`.

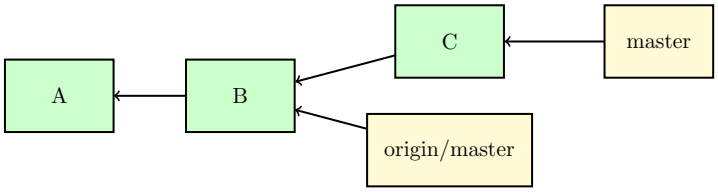
Nuestro clon del repositorio destacado incluirá, además de los objetos que en ese momento tenía el repositorio, referencias de sólo lectura a las revisiones en que se hallaban las ramas disponibles. Estas últimas son conocidas como *ramas remotas*, y para desarrollar a partir de ellas necesitaremos usar nuestras propias ramas locales: podemos comenzar nuevas ramas desde los puntos señalados por las ramas remotas, o reunir nuestras ramas locales con ellas.

1.2. Recepción de revisiones

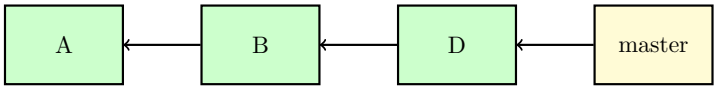
Periódicamente, podremos tomar los nuevos objetos que se hayan creado en cualquiera de los repositorios remotos que estemos monitorizando y añadirlos a nuestro repositorio, actualizando las ramas remotas. Para ello, usaremos la orden `git fetch [repositorio [refspec]]`. Según vayamos dando más o menos argumentos, su lógica cambiará:

- Sin argumentos, es equivalente a `git fetch origin`.
- Si se especifica sólo el repositorio, mediante su URL o el nombre de su *remote*, buscará su valor en varios sitios. Uno de ellos es la variable *remote.mirama.fetch* de `.git/config`, donde *mirama* es el nombre de la rama actual.

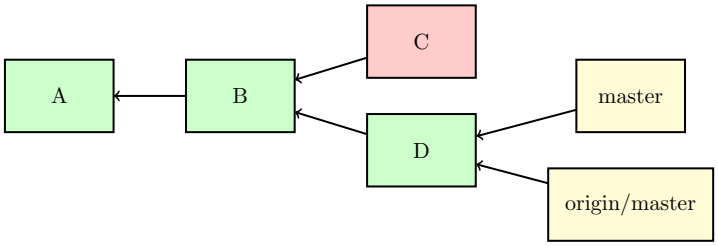
  Git aprovecha este comportamiento cuando clonamos un repositorio: al dar el valor correcto a la variable *remote.origin.fetch*, podemos hacer `git fetch` sin tener que pasar ningún argumento.
- Con los dos argumentos su comportamiento ya queda completamente definido. El segundo argumento es un tanto especial: la sintaxis es algo más compleja, teniendo la forma `[+]fuente[:destino]`. Su significado también se va construyendo poco a poco:
  - Con `fuente` a secas, decimos que queremos recibir la rama remota `fuente` y situar la punta en la referencia `FETCH_HEAD`. Esto es útil



(a) Local



(b) Remoto



(c) Local tras `git fetch origin +master:master`: se pierde `C`

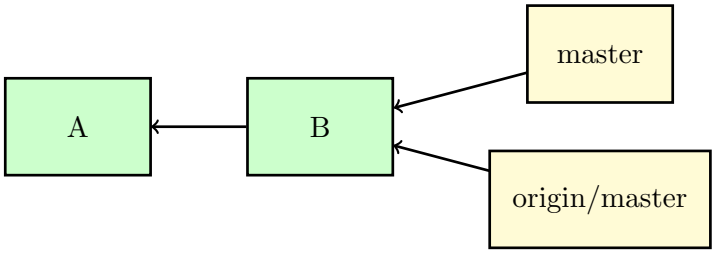
Figura 1: Problema introducido por actualizaciones no *fast forward*

sobre todo para ver qué cambios se han introducido en una rama remota sin que afecte a nuestro repositorio.

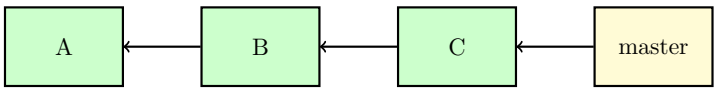
- `fuente:destino` toma la referencia fuente y actualiza o crea con ella la referencia destino. `refs/heads/*:refs/remotes/origin/*`, por ejemplo, es el valor configurado en *remote.origin.fetch* al clonar un repositorio. Este valor es el que indica a `git fetch` que actualice todas las ramas remotas con sus valores en el repositorio remoto, manteniendo los nombres.
- `+fuente:destino`, por otro lado, permite que aunque la punta de la fuente no sea descendiente de la punta actual del destino (es decir, no se pueda simplemente hacer un *fast forward*), se permita la actualización. Esta opción conlleva ciertos riesgos: puede verse en la figura 1 cómo forzar la actualización ha hecho que `C` quede sin referenciar por ninguna rama.

1.3. Reunión de las ramas locales con las remotas

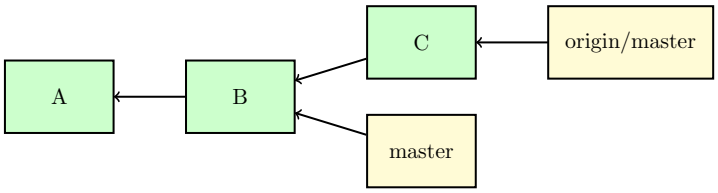
`git fetch` no reúne nuestras ramas locales con las ramas remotas (véase la figura 2 en la página siguiente): para ello tendremos que usar `git merge`, tal y como se vio en los apuntes de ramas. Podemos unir `git fetch` y `git merge` en una sola orden, `git pull [repositorio refspecs]`.



(a) Local



(b) Remoto



(c) Local tras `git fetch`: `master` no se mueve

Figura 2: Ejemplo de cómo `git fetch` no reúne ramas

En su forma completa con la referencia al repositorio y al menos el nombre de una rama de dicho repositorio, el proceso seguido por `git pull X Y...` es:

1. Primero se ejecuta `git fetch` con todos los argumentos recibidos para recibir los objetos del repositorio remoto.

2. Por último se reúne la rama actual con las puntas de todas las ramas referenciadas en los argumentos `Y...`, usando `git merge`. Si se pasa la opción `--rebase` se sustituye por `git rebase`. No olvidar que sólo puede usarse esta última orden si aún nadie tiene la parte del historial que va a ser afectada.

Cuando no decimos nada, `git pull` utiliza algunos ficheros de configuración, con ciertos cambios. Simplificando un poco, sería algo así:

1. La referencia al repositorio en cuestión se obtiene a partir de la variable local *branch.mirama.remote* de `.git/config`, suponiendo que nos hallamos en la rama `mirama`.

2. Los argumentos para `git fetch` se extraen de la variable local *remote.miremoto.fetch*, suponiendo que el remoto antes indicado era «miremoto».

3. Las ramas del repositorio remoto con que se debería reunir la rama actual se encuentran en la variable local *branch.mirama.merge*.

Estas tres variables son configuradas automáticamente para la rama principal y el *remote* «origin» creados tras un clonado. Si creamos una nueva rama local, tendremos que indicarle de qué *remote* y rama remota actualizarse. Suponiendo que queramos desarrollar a partir de la rama `mirama` del *remote* «miremoto» en una nueva rama `nuevarama`, tenemos varias opciones:

1. Ir indicándolo manualmente con más argumentos:

```
git pull miremoto mirama:nuevarama
```

2. Crear la rama con la opción `--track`. Hay varias formas de hacerlo. La más sencilla es:

```
git checkout −b nuevarama −−track miremoto/mirama
```

   Se trata de un atajo para:

```
git branch −−track nuevarama miremoto/mirama
git checkout nuevarama
```

3. Configurarlo manualmente. Hay varios posibles ficheros a modificar, pero si usáramos `.git/config`, por ejemplo, ejecutaríamos algo así:

```
git checkout −b nuevarama miremoto/mirama
git config branch.nuevarama.remote miremoto
git config branch.nuevarama.merge mirama
```

1.4. Envío de revisiones

Otras veces desearemos enviar nuestras propias revisiones a otro repositorio. Esta acción se realiza mediante la orden `git push [repositorio [refspecs]]`, y al igual que `git fetch`, su significado se va construyendo progresivamente a medida que se le pasan argumentos:

- `git push` sin argumentos equivale a `git push origin`.
- Si se ejecuta `git push nombre`, y existe una variable *remote.nombre.push* con una o más *refspecs* `Y...`, entonces la orden equivale a `git push nombre Y...`.
- Si se ejecuta `git push nombre` pero no existe la anterior variable, entonces equivale a `git push nombre :`.

- Una vez todos los argumentos quedan determinados, `git push X Y...` actualizaría cada una de las ramas destino con los objetos del repositorio local, recorriendo cada uno de los *refspec* proporcionados.

  Estos *refspec* son un tanto especiales, y difieren en bastantes cosas de los de `git fetch` (y por tanto de los de `git pull`). La sintaxis básica `[+]fuente[:destino]` sigue siendo admitida y funciona de la misma forma, pero hay algunas diferencias:

  - Si no damos el nombre del destino, se asume que es el mismo que el origen. Estas dos órdenes son equivalentes:

    ```
    git push origin foo
    git push origin foo:foo
    ```

  - La fuente puede ser cualquier *commitish*, y en particular podemos enviar etiquetas:

    ```
    git tag -a v1.0
    git push origin v1.0
    ```

    Si usamos un *commitish* a una revisión cualquiera, estaremos obligados a dar un destino:

    ```
    git push origin master~5:otrarama
    ```

  - Podemos omitir la fuente y dejar sólo el destino, borrando así la rama (`:mirama`)o etiqueta (`:v1.0`) indicada en el destino.
  - Podemos omitir tanto la fuente como el destino, quedando `:`, o `+:` para actualizar incluso cuando no sea un *fast forward*. El comportamiento de `git push` al recibir este argumento es un tanto peculiar: básicamente, cada rama en el repositorio indicado es actualizada desde la rama local con el mismo nombre, si existe.

1.5. Autoría de las líneas de un fichero

Para ver quién es el responsable de cada línea de un fichero, es decir, la última persona que la modificó, y cuándo se hizo, Git nos proporciona la orden `git blame`, a la que en su forma más sencilla únicamente tenemos que pasar la ruta de un fichero:

```
git blame fichero
```

Si queremos activar todas las heurísticas de líneas movidas (`-M`) y copiadas (`-C`) desde otros ficheros al máximo (otra vez `-C`), mostrando los números de línea originales (`-n`), usaremos:

```
git blame -n -M -C -C fichero
```

Evidentemente, esta segunda versión puede que tarde más, pero nos dará mucha más información.

2. Conectividad por red entre repositorios

2.1. Introducción

Git puede trabajar completamente en local, utilizando rutas a otros directorios y URL de la forma `file:///ruta/a/.git`, pero esto evidentemente no es suficiente: normalmente, los repositorios con los que querremos colaborar se hallarán en otras máquinas, y para ello tendremos que utilizar alguna conexión de red.

Git dispone de múltiples opciones según nuestras necesidades, e incluso facilita su uso entre entornos donde la conectividad es intermitente o inexistente o no tenemos los permisos de escritura necesarios. En esta sección daremos un repaso por las distintas opciones, con sus ventajas y desventajas.

Antes que nada, es importante destacar que cualquier repositorio que sea accesible por terceras partes debería ser lo que se conoce en Git como un repositorio «pelado» (*bare* en el original inglés). Estos repositorios tienen la particularidad de que no disponen de un directorio de trabajo ni un índice, y así evitan que cualquiera trabaje en ellos y se confunda cuando cambien sus objetos tras un `git push` remoto y no se actualice su directorio de trabajo. Sólo sirven como punto de reunión para varios desarrolladores. Lo normal es que, en caso de que necesitemos un repositorio público, nuestro repositorio privado sea un clon de éste, para facilitarnos posteriormente el intercambio de revisiones.

2.2. Acceso por SSH

El acceso por el protocolo cifrado y autenticado Secure Shell (SSH) es el preferido, de lejos, para realizar cualquier tipo de operación con Git que requiera autenticación: es el utilizado normalmente para realizar la operación `git push` y también puede usarse para limitar el acceso de lectura. Las URL de acceso a un repositorio por SSH tienen la forma `usuario@host:ruta`, del mismo estilo que se usan en herramientas como `scp`, por ejemplo.

En su forma más básica, este método de acceso requiere que cada usuario tenga una cuenta en una máquina dedicada. De todas formas, Git incluye un shell de entrada limitado exclusivamente a las órdenes `git push` y `git fetch` (y por ende `git pull`) llamado `git-shell`, con lo que se reduce a un problema sobre todo administrativo, y no tanto de seguridad. Podemos incluso reunir todos los directorios personales en uno solo y controlar el acceso a cada repositorio mediante el sistema de permisos clásico de UNIX.

Hay otra posibilidad que puede resultar más cómoda y a la vez más segura de utilizar: Gitosis. Se trata de un conjunto de guiones Python que permiten gestionar una serie de repositorios Git a través de un repositorio Git. El proceso general, tomado de [2], es el que sigue:

1. Necesitaremos algunos paquetes Debian:

```
sudo aptitude install python-setuptools
```

2. Descargamos Gitosis clonando su repositorio, y lo instalamos:

```
git clone git://eagain.net/gitosis.git
cd gitosis
sudo python setup.py install
```

3. Se crea una única cuenta de usuario llamada «git», que no tendrá contraseña. Esta cuenta se trata de un usuario de sistema con su propio directorio personal y un shell Bourne básico (`/bin/sh`).

4. Crearemos un par de claves pública y privada SSH ejecutando:

```
ssh-keygen -t rsa
```

5. Copiaremos la clave pública que se habrá creado en su localización a `/tmp`, y ejecutaremos:

```
sudo -H -u git gitosis-init < /tmp/id_rsa.pub
```

   Entre otras cosas, esta acción prepara el directorio personal del usuario de Git para que sólo sea accesible para el usuario con la clave pública SSH proporcionada, y únicamente para hacer `git push`, `git pull` `git fetch`. Para ello configura debidamente el fichero `~/.ssh/authorized_keys`

6. Ya podemos clonar el repositorio de administración de Gitosis:

```
git clone git@127.0.0.1:gitosis-admin.git
```

   Si la clave privada SSH se halla cifrada, se nos pedirá la clave. Así, un atacante tendría no sólo que tener una copia de la clave privada, sino también su propia clave de cifrado. Además, normalmente las claves privadas SSH no viajan por la red (tenemos una por usuario y máquina).

7. El contenido del repositorio es muy sencillo: hay un directorio `keydir` en el que situaremos todas las claves públicas SSH de los usuarios que queramos reconocer (con extensión `.pub`), y un fichero `gitosis.conf` que modificaremos para definir nuevos grupos de usuarios y cambiar las opciones por defecto y/o específicas a ciertos repositorios. Sólo tendremos que ir creando nuevas revisiones de la configuración y empujarlas al servidor.

   Podemos seguir las instrucciones de la web anterior, o las del fichero `example.conf` que incluye Gitosis. En particular, el segundo fichero

menciona la posibilidad de dejar que Gitosis sea quien controle el acceso mediante Web o con el demonio de lectura anónima a ciertos repositorios.

2.3. Acceso por el protocolo Git

El acceso por SSH es bastante eficiente en términos de ancho de banda, pero evidentemente el cifrado, la autenticación y demás imponen una carga administrativa y de rendimiento. Cuando no necesitemos el cifrado ni la autenticación, como en el caso del acceso de sólo lectura en los proyectos de software libre y/o código abierto, podemos usar el protocolo de transferencia propio de Git, que es extremadamente eficiente y permite clonar de manera rápida los repositorios. Las URL tienen la forma `git://host/ruta`.

Para usarlo para compartir nuestros repositorios, únicamente tenemos que iniciar el demonio, que quedará normalmente a la escucha en el puerto 9418 TCP, con una lista de rutas a los directorios `.git` o, preferiblemente, a los repositorios «pelados»:

```
git daemon /ruta/a/pelado.git /ruta/a/nopelado/.git
```

Por razones de seguridad, el demonio se negará en redondo a servir cualquier otra cosa que no sea un repositorio Git, e incluso así, sólo lo hará si contiene un fichero (da igual que esté vacío) llamado `git-daemon-export-ok`. Esta última comprobación puede desactivarse con `--export-all` si se desea. Si utilizamos Gitosis, con poner la variable *daemon* al valor «yes» y empujar la revisión correspondiente se nos creará sin más problemas.

Otra configuración muy común es situar todos los repositorios como subdirectorios de un directorio principal (como en el caso de Gitosis). Para este escenario, el demonio dispone de la orden `--base-path`. Si estuviéramos usando el esquema propuesto por Gitosis, podríamos hacer algo así:

```
sudo -H -u git git daemon --base-path=/home/git/repositories/
```

El protocolo de Git permite hacer envíos anónimos con `git push`, pero no por defecto: tendremos que invocar al demonio y activar dicho servicio explícitamente con `--enable=receive-pack`. Sólo se recomienda su uso en entornos muy amigables, como una LAN muy controlada.

2.4. Acceso por HTTP(S)

Si nuestra mayor preocupación es el mantenimiento de conectividad incluso ante cortafuegos muy restrictivos, muy comunes en las grandes organizaciones, nuestra mejor opción usar HTTP o HTTPS, que son los únicos protocolos permitidos por la mayoría.

La configuración de sólo lectura de Git para HTTP o HTTPS es muy sencilla: basta con ofrecer el repositorio «bare» por Apache y activar el «hook»

`post-update` que viene de ejemplo en el código fuente de Git. En esta configuración, los envíos se seguirán haciendo a través de SSH.

Sin embargo, si se quiere dar acceso de escritura, hay dos opciones: la más antigua opera a través de WebDAV, y la más reciente (desde Git 1.7.0) se conoce como «smart HTTP» y opera a través de un CGI proporcionado por Git. Hoy en día se recomienda evitar el acceso por WebDAV y usar sólo la opción de «smart HTTP». De todos modos, veremos ambos casos.

2.4.1. Acceso por HTTP(S) a través de WebDAV

Antes de empezar, es importante notar que esta opción tiene problemas de rendimiento, requiere un trabajo `cron` especial, no garantiza que las revisiones se registren de forma atómica y puede tener problemas de integridad si el cliente utiliza versiones con fallos de `libcurl`. Si de todos modos se desea ir por esta vía, habría que seguir estos pasos:

1. Debería de ser un repositorio «pelado», como de costumbre.
2. Hay que ejecutar manualmente una vez al menos por repositorio la orden `git-update-server-info`, que dejará información necesaria para que el cliente pueda seguir funcionando incluso con un servidor que sólo sabe servir ficheros estáticos.
3. Por si alguien realiza algún `git push` posteriormente por SSH, se tendrá que dar permisos de ejecución al gancho `hooks/post-update` del repositorio, para que se ejecute `git-update-server-info` en cada actualización de nuevo.
4. Habrá que configurar en el fichero global para Git para el usuario o usuarios que alojen los repositorios Git que no se empaqueten las referencias de los repositorios (suponemos que sólo servimos «pelados» aquí), cosa que haría dejar de funcionar a los clientes por HTTP:

   ```
   git config --global gc.packrefs notbare
   ```

Hecho esto, servirlo para sólo lectura es tan sencillo como ofrecer sus ficheros de manera estática por cualquier servidor HTTP que queramos. Si además queremos poder hacer `git push`, habrá que seguir algunas recomendaciones adicionales (complementadas y ampliadas en [6]):

1. Habrá que activar su acceso por WebDAV, habilitando el sistema de cerrojos necesario y restringiendo su acceso a los usuarios autenticados. Se incluye en el repositorio de estos apuntes un fichero `conf-apache/gitweb` de configuración de Apache de ejemplo. Este fichero también detalla una forma en que se podría utilizar el CGI de Gitweb para servir una interfaz de sólo lectura al contenido de una serie de repositorios.

2. Además, se tendrá que asegurar que todos los clientes utilicen versiones relativamente recientes de Git (1.5.4+ como mínimo, aunque se recomienda 1.5.6+) y de la biblioteca Curl (supuestamente 7.6+ vale, pero nos hemos encontrado con problemas, así que recomendamos 7.8+). Esto se debe a que, como no hay servidor que se ocupe de actualizar los ficheros del repositorio por su cuenta, son los propios clientes (con los defectos que traigan sus versiones) los que lo actualizan todo, sustituyendo entre otras cosas a las ejecuciones posteriores a la primera del guión `git-update-server-info`.

2.4.2. Acceso por HTTP(S) a través de CGI

La mejor forma de acceder a un repositorio Git a través de HTTP(S) es mediante el CGI `git-http-backend` que viene incluido con Git. Este CGI ofrece sólo dos servicios: uno para subir cambios y otro para descargar código. En ambos casos, sólo se requiere una petición HTTP para cada orden que lancemos desde Git, por lo que el rendimiento es muy similar al de SSH, y tenemos garantizada la atomicidad de las revisiones.

En un Apache con los módulos `cgi`, `alias` y `env` activados, asumiendo que los repositorios estén bajo `/srv/git` (escribibles por el usuario de Apache) y que `git-http-backend` está en la ruta por omisión en Ubuntu, se podría usar una configuración como:

```
SetEnv GIT_PROJECT_ROOT /srv/git/
SetEnv GIT_HTTP_EXPORT_ALL
ScriptAlias /git/ /usr/libexec/git-core/git-http-backend/
<Location /git>
  Order allow,deny
  Allow from all

  AuthType Basic
  AuthName Git
  Require valid-user
  ...
</Location>
```

La base de datos de usuarios puede ser un `htpasswd` normal y corriente de Apache, o podemos integrar un manejador de autenticación personalizado. En el caso de Redmine se utilizan dos módulos Perl que proporciona el mismo proyecto.

Sólo hay una pequeña incomodidad, y es que Git nos preguntará todo el rato por el usuario y contraseña. Para evitar esto, en GNU/Linux se pueden indicar estos datos en el fichero `~/.netrc`, que sólo debería ser legible por nuestra cuenta de usuario:

```
touch ~/.netrc
chmod 600 ~/.netrc
```

```
echo −e "machine host\nlogin usuario\npassword clave" > ~/.netrc
```

2.5. Acceso por Web

Aunque técnicamente sigue siendo HTTP, consideramos que el acceso mediante páginas Web constituye su propio apartado. Para un solo directorio, podemos utilizar `git instaweb`, para ver cómo quedaría Gitweb sobre él. Suponiendo que utilizamos Apache, nos situaríamos en nuestro repositorio de trabajo y ejecutaríamos algo así:

```
git instaweb −−httpd=apache
```

Para más de un directorio, se recomienda tener un servidor Web debidamente configurado, que utilice el CGI de Gitweb tras haberlo configurado debidamente. Si usamos Gitosis y sus variables *gitweb*, no debemos de olvidar referenciar en `gitweb.cgi` a la lista de proyectos incluida en `~git/gitosis/projects.list`, o dichas variables no tendrán efecto alguno.

Otra opción es usar algún software de gestión de proyectos y la integración de Git que incluya. Existe un plug-in, aunque algo inmaduro, de Git para el conocido Trac [5, 7], pero recomiendo en su lugar utilizar Redmine [4], que además de tener soporte para una mayor variedad de sistemas de control de versiones, permite utilizar la misma base de datos para gestionar múltiples proyectos.

La única pega es que la configuración inicial es algo más compleja, recomendándose el uso de Apache como proxy inverso del servidor Apache Mongrels que alojará a Redmine, ya que es una aplicación Ruby. El proceso de instalación se halla bien documentado en su página oficial, y de todas formas se ha incluido también en `conf-apache/redmine-git` un fichero de configuración de Apache que representa un caso más realista.

2.6. Acceso en conectividad limitada

Puede que no dispongamos de una conexión de red decente al repositorio al que o desde el cual queramos enviar nuestros cambios: la red no funciona, tenemos poco ancho de banda, el cortafuegos es más estricto de la cuenta o estamos de viaje, por ejemplo. Otras veces, puede que sencillamente no tengamos los permisos necesarios de escritura, pero aún así queramos hacer lo más fácil posible a los desarrolladores originales el uso de nuestras aportaciones.

Para estos casos, Git dispone de ciertas facilidades:

- `git bundle` permite empaquetar en un único fichero una serie de revisiones de una rama, llevarlo en el medio que haga falta a otro sitio (como un *pendrive* USB o un CD, por ejemplo), y volver a crear esas revisiones en el otro lado. Los pasos serían:

1. Creamos el fichero con:

```
git bundle create mibundle a..b
```

   Es importante que `a` sea el *commitish* de una revisión que sepamos que existe en el otro lado, y que `b` sea alguna rama que exista también en el otro lado, o de lo contrario no podremos realmente actualizar nada.

2. Lo llevamos como sea a donde sea y lo usamos como sustituto del repositorio de un `git fetch` o `git pull`:

```
git pull ../mibundle master
git fetch ../mibundle master:origin/master
```

   La primera orden es un `git pull` normal y corriente, y la segunda orden actualiza una rama remota a partir de un *bundle*.

- Si lo que nos falta es acceso de escritura a un repositorio, la forma más directa es crear un fichero `diff` y que otra persona lo aplique. No hace falta que usemos Git para la primera parte, pero `git diff a..b` puede ser muy útil, como aquí:

```
git diff HEAD^..HEAD | gzip -9 > miparche.diff.gz
```

  Ahora se envía el fichero `miparche.diff.gz` a quien sea, que ejecutaría esto sobre su repositorio:

```
zcat miparche.diff.gz | git apply -
```

- Otra forma más directa de hacerlo sería dar formato a cada parche como un correo especial que pudiera recibirse y aplicarse directamente. Esto se puede hacer con las órdenes `git format-patch`, `git send-email` y `git am`, que hacen lo siguiente:

  1. `git format-patch` convierte una serie de revisiones a un formato que es legible por humanos y contiene toda la información necesaria para Git. Normalmente se vuelcan como ficheros de un directorio de correos:

```
git format-patch -o parches master~2..
```

  2. Ahora se envían por correo:

```
git send-email ../parches
```

  3. Si recibimos los correos en formato *mbox*, usando `mutt`, por ejemplo, podemos aplicar sus cambios con:

```
git am ~/mbox
```

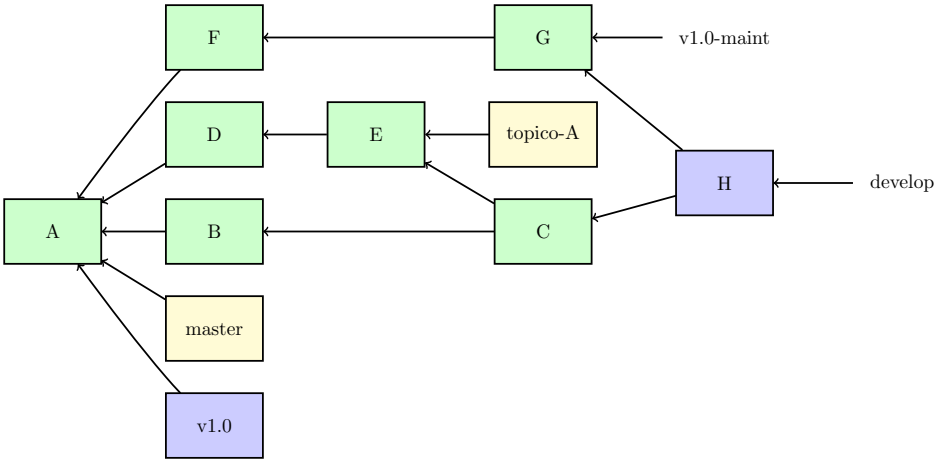


Figura 3: Ramificación usual de un repositorio Git

3. Flujos de trabajo usuales con Git

3.1. Introducción

Esta sección describirá algunos de los flujos de trabajo más usuales con Git. Git, al tratarse de un sistema distribuido de control de versiones, no nos impone ninguno en particular. Sin embargo, en la práctica se suelen adoptar alguno de los tres enfoques que veremos a continuación. Estos enfoques han sido tomados de la excelente presentación «Getting Git» de Scott Chacon [1].

En la mayoría de los repositorios Git, existe una rama `master` que únicamente registra las revisiones estables (publicables) del sistema, y otra rama `develop` sobre la cual se van situando las revisiones inestables (`next` en el repositorio de Git). Las ramas que implementan ciertas funcionalidades salen y se reúnen después con `develop`, y cuando `develop` está lo bastante estable, se reúne con `master` (véase la figura). También pueden aparecer ramas de mantenimiento de versiones más antiguas, como `maint` en el repositorio de Git. Se muestra un ejemplo del aspecto típico de un repositorio Git en funcionamiento en la figura 3, donde los `git push` se representan con arcos rojos y los `git pull` con arcos azules.

3.2. Centralizado

El flujo centralizado se corresponde con el utilizado en herramientas como Subversion o CVS, y utiliza un subconjunto muy pequeño de la funcionalidad de Git. Se tiene un repositorio «pelado» como punto central de reunión del trabajo de todos los demás desarrolladores, que clonan ese repositorio (véase la figura 4 en la página siguiente).

La forma normal de trabajar es:

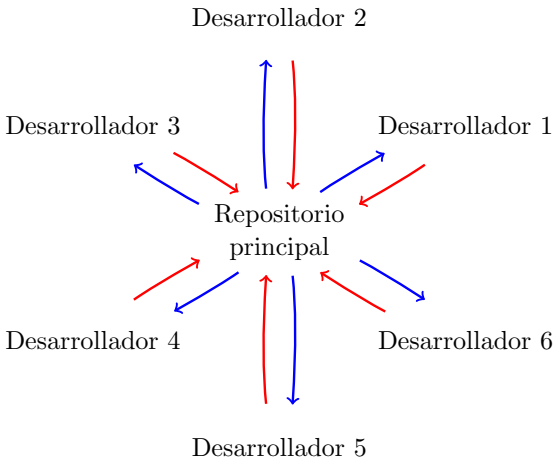


Figura 4: Flujo centralizado de trabajo

1. Si no se tiene el repositorio ya, habrá que clonarlo, y crear una rama local `develop` que vigile la rama remota del mismo nombre:

```
git clone url-a-repositorio
cd repositorio
git checkout -b develop --track origin/develop
```

2. Antes de empezar, obtenemos los últimos cambios sobre `develop`:

```
git checkout develop
git pull
```

3. Se trabaja a partir de una rama local propia desde la rama remota de desarrollo:

```
git checkout -b mi-funcionalidad --track origin/develop
```

4. Editamos, depuramos, etc. De vez en cuanto nos interesará reunirnos con el trabajo de los demás:

```
git pull --rebase
```

5. Una vez hayamos terminado nuestro trabajo, cambiaremos a la rama local de desarrollo, la actualizaremos y la reuniremos con nuestra rama local:

```
git checkout develop
git pull
git merge mi-funcionalidad
```

Si hemos estado reuniendo con frecuencia nuestro trabajo con el de los demás, no deberían de producirse conflictos en este paso.

6. Ahora podemos eliminar la antigua rama, si así lo deseamos, y empujar los cambios al repositorio:

```
git branch −d mi−funcionalidad
git push
```

7. Cuando el director de proyecto (o el responsable de integración) vea que la rama de desarrollo tiene la suficiente calidad como para constituir la próxima publicación, puede actualizar sus dos ramas `master` y `develop`, reunirlas, etiquetar y firmar la versión por GnuPG y empujar los cambios:

```
git checkout develop
git pull
git checkout master
git pull
git merge develop
git tag −s v1.0
git push −−tags
```

3.3. Distribuido

La verdadera utilidad de Git es como un sistema distribuido, en el que no existe una autoridad central de por sí. En ese aspecto ya no hay mucho que podamos restringir, y depende ya del tipo de proyecto en que nos hallemos.

De todas formas, generalmente existe algún repositorio destacado o «bendecido» que la mayoría de los usuarios y desarrolladores toman como referencia, y que tiene el acceso de escritura restringido a unos pocos que se ocupan fundamentalmente de integrar el trabajo de los repositorios públicos de los demás desarrolladores (como en el propio repositorio de Git). Los desarrolladores actualizan cuando lo ven necesario sus repositorios públicos a partir de sus privados usando `git push`. Este modelo con un único nivel de integración se conoce como el «modelo del integrador», y puede verse un esquema en la figura 5 en la página siguiente. Es interesante ver en los arcos de `git pull` (azules) del repositorio público del desarrollador 2 al privado del 1 y viceversa cómo los desarrolladores pueden colaborar entre sí directamente sin necesidad de pasar por el repositorio bendecido.

Cuando el trabajo de integración ya no puede ser realizado por una única persona, surge el «modelo del dictador y sus tenientes», en el que el trabajo de integración se hace en dos niveles: el dictador integra el trabajo de sus tenientes, y los tenientes el de los desarrolladores «de a pie». Es el caso del kernel Linux. Se puede ver un ejemplo en el que se han reunido en un solo

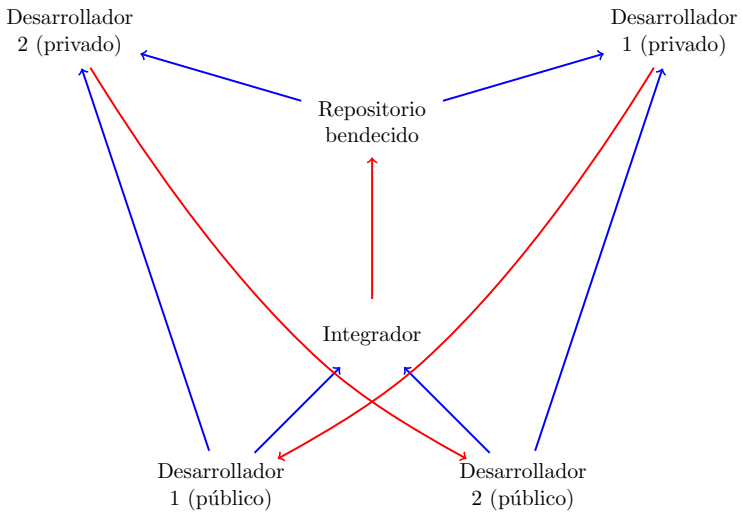


Figura 5: Flujo del integrador

nodo del grafo el repositorio público y privado de cada desarrollador en la figura .

En cuanto a las órdenes necesarias para su uso, no hay mucho que decir:

1. Primero clonaríamos el repositorio bendecido a un repositorio público «pelado»:

   ```
   git clone --bare url-repositorio-bendecido
   ```

2. Ahora clonaríamos nuestro repositorio público a otro privado, esta vez con su directorio de trabajo:

   ```
   git clone url-o-ruta-repositorio-publico ruta-repositorio-privado
   ```

3. Nos introduciríamos en el repositorio e iríamos trabajando de la forma usual. Si queremos colaborar con alguien, podemos añadir la URL de su repositorio como un *remote* más ejecutando:

   ```
   git remote add nombre-remote url-remote
   ```

   Una vez queda añadido, podemos hacer varias cosas:

   - Ver todos los disponibles con `git remote`.
   - Ver los detalles de uno en particular con `git remote show nombre-remoto`.
   - Retirar uno de la lista con `git remote rm nombre-remoto`.
   - Actualizar las ramas remotas de todos los *remotes* con `git remote update`.

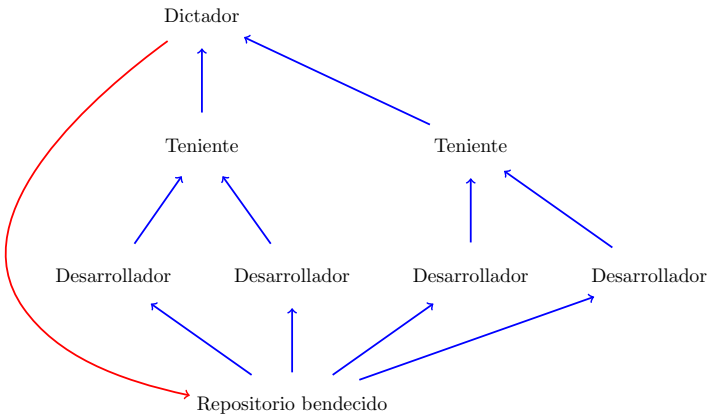


Figura 6: Flujo del dictador y sus tenientes

- Eliminar las ramas remotas que ya no existan en sus repositorios con `git remote prune`.

Por ejemplo, si queremos añadir al repositorio bendecido como otro *remote* más en el repositorio privado, podríamos hacer:

```
git remote add central url-del-central
```

Así, podemos actualizar nuestra rama principal desde la principal del repositorio bendecido con:

```
git checkout master
git pull central master
```

Referencias

[1] Scott Chacon. *RailsConf Git Talk*. `http://www.gitcasts.com/posts/railsconf-git-talk`, junio 2008.

[2] Garry Dolley. *Hosting Git repositories, The Easy (and Secure) Way*. `http://scie.nti.st/2007/11/14/hosting-git-repositories-the-easy-and-secure-way`, noviembre 2007.

[3] *Redmine - Overview*. `http://www.redmine.org`, 2008.

[4] Herbert Valerio Riedel y Hans Petter Jansson. *Track Hacks - GitPlugin*. `http://trac-hacks.org/wiki/GitPlugin`, febrero 2008.

[5] Johannes Schindelin, Rutger Nijlunsing, y Matthieu Moy. *Setting up a Git repository which can be pushed into and pulled from over*

HTTP(S). http://www.kernel.org/pub/software/scm/git/docs/howto/setup-git-server-over-http.txt, agosto 2006.

[6] *The Trac Project*. http://trac.edgewall.org/, 2008.