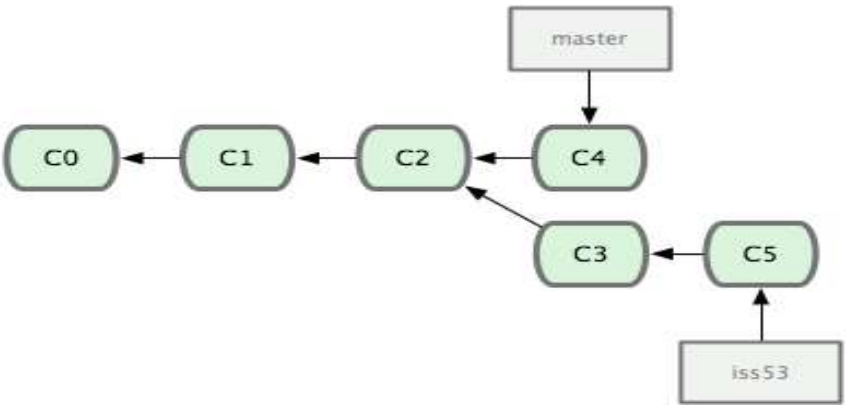
Git

Ramificaciones



Contenido

Ramificaciones en Git

Cualquier sistema de control de versiones moderno tiene algún mecanismo para soportar distintos ramales. Cuando hablamos de ramificaciones, significa que tu has tomado la rama principal de desarrollo (master) y a partir de ahí has continuado trabajando sin seguir la rama principal de desarrollo. En muchas sistemas de control de versiones este proceso es costoso, pues a menudo requiere crear una nueva copia del código, lo cual puede tomar mucho tiempo cuando se trata de proyectos grandes.

Algunas personas resaltan que uno de los puntos mas fuertes de Git es su sistema de ramificaciones y lo cierto es

que esto le hace resaltar sobre los otros sistemas de control de versiones. ¿Porqué esto es tan importante? La forma en la que Git maneja las ramificaciones es increíblemente rápida, haciendo así de las operaciones de ramificación algo casi instantáneo, al igual que el avance o el retroceso entre distintas ramas, lo cual también es tremendamente rápido. A diferencia de otros sistemas de control de versiones, Git promueve un ciclo de desarrollo donde las ramas se crean y se unen ramas entre sí, incluso varias veces en el mismo día. Entender y manejar esta opción te proporciona una poderosa y exclusiva herramienta que puede, literalmente, cambiar la forma en la que desarrollas.

.1¿Qué es una rama?

Para entender realmente cómo ramifica Git, previamente hemos de examinar la forma en que almacena sus datos. Recordando lo citado en el capítulo 1, Git no los almacena de forma incremental (guardando solo diferencias), sino que los almacena como una serie de instantáneas (copias puntuales de los archivos completos, tal y como se encuentran en ese momento).

En cada confirmación de cambios (commit), Git almacena un punto de control que conserva: un apuntador a la copia puntual de los contenidos preparados (staged), unos metadatos con el autor y el mensaje explicativo, y uno o varios apuntadores a las confirmaciones (commit) que sean padres directos de esta (un padre en los casos de confirmación normal, y

múltiples padres en los casos de estar confirmando una fusión (merge) de dos o mas ramas).

Para ilustrar esto, vamos a suponer, por ejemplo, que tienes una carpeta con tres archivos, que preparas (stage) todos ellos y los confirmas (commit). Al preparar los archivos, Git realiza una suma de control de cada uno de ellos (un resumen SHA-1, tal y como se mencionaba en el capítulo 1), almacena una copia de cada uno en el repositorio (estas copias se denominan "blobs"), y guarda cada suma de control en el área de preparación (staging area):

```
$ git add README test.rb LICENSE
$ git commit -m 'initial commit of my project'
```

Cuando creas una confirmación con el comando `git commit`, Git realiza sumas de control de cada subcarpeta (en el ejemplo, solamente tenemos la carpeta principal del proyecto), y las guarda

como objetos árbol en el repositorio Git. Después, Git crea un objeto de confirmación con los metadatos pertinentes y un apuntador al objeto árbol raiz del proyecto. Esto permitirá poder regenerar posteriormente dicha instantánea cuando sea necesario.

En este momento, el repositorio de Git contendrá cinco objetos: un "blob" para cada uno de los tres archivos, un árbol con la lista de contenidos de la carpeta (más sus respectivas relaciones con los "blobs"), y una confirmación de cambios (commit) apuntando a la raiz de ese árbol y conteniendo el resto de metadatos pertinentes. Conceptualmente, el contenido del repositorio Git será algo parecido a la Figura 3-1

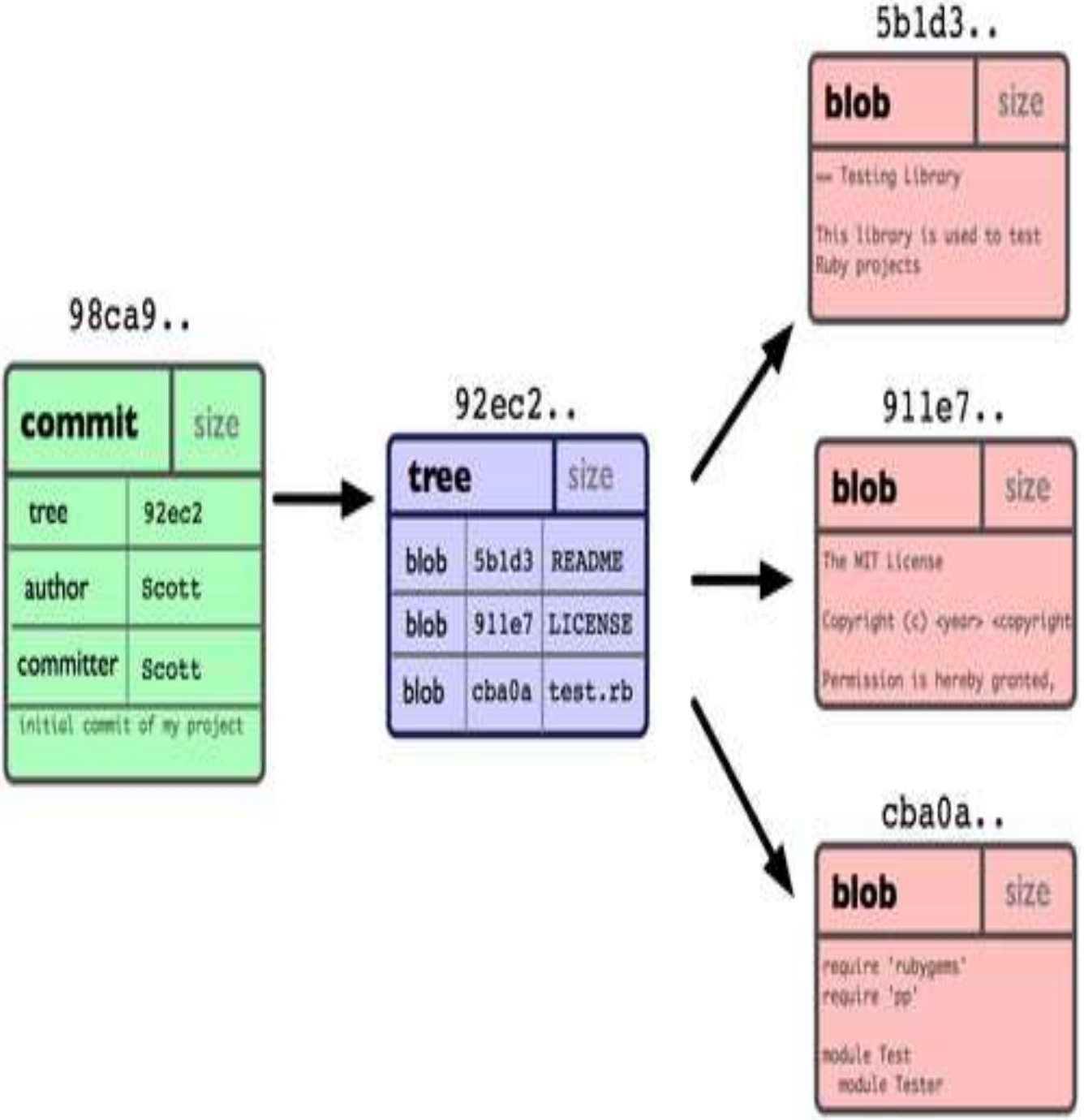


Figura 3-1. Datos en el repositorio tras una confirmación sencilla.

Si haces más cambios y vuelves a confirmar, la siguiente confirmación guardará un apuntador a esta su confirmación precedente. Tras un par de confirmaciones más, el registro ha de ser algo parecido a la Figura 3-2.

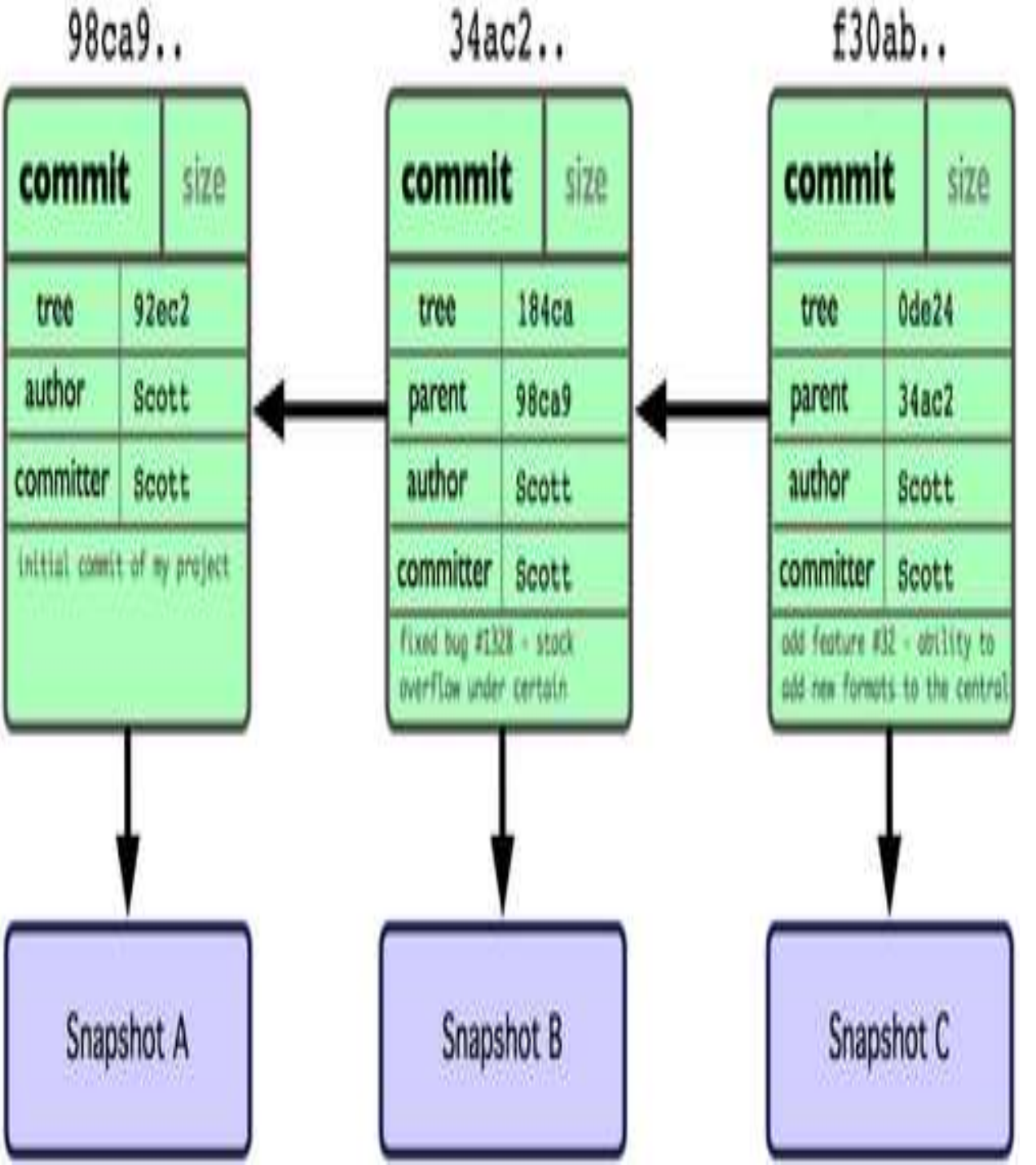


Figura 3-2. Datos en el repositorio tras una serie de confirmaciones.

Una rama Git es simplemente un apuntador móvil apuntando a una de esas confirmaciones. La rama por defecto de Git es la rama `master`. Con la primera confirmación de cambios que realicemos, se creará esta rama principal `master` apuntando a dicha confirmación. En cada confirmación de cambios que realicemos, la rama irá avanzando automáticamente. Y la rama `master` apuntará siempre a la última confirmación realizada.

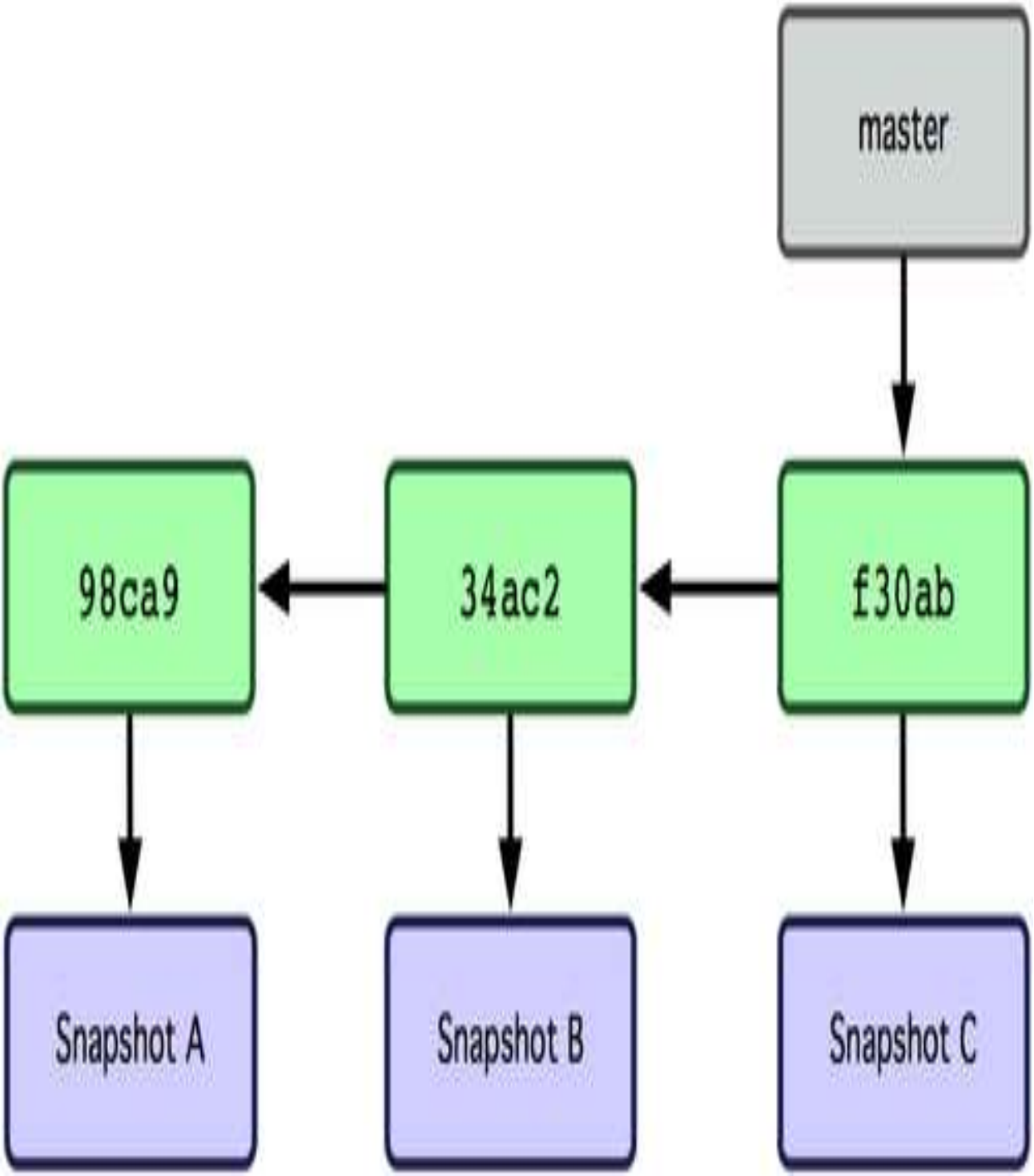


Figura 3-3. Apuntadores en el registro de confirmaciones de una rama.

¿Qué sucede cuando creas una nueva rama? Bueno..., simplemente se crea un nuevo apuntador para que lo puedas mover libremente. Por ejemplo, si quieres crear una nueva rama denominada "testing". Usarás el comando git branch:

```
$ git branch testing
```

Esto creará un nuevo apuntador apuntando a la misma confirmación donde estés actualmente (ver Figura 3-4).

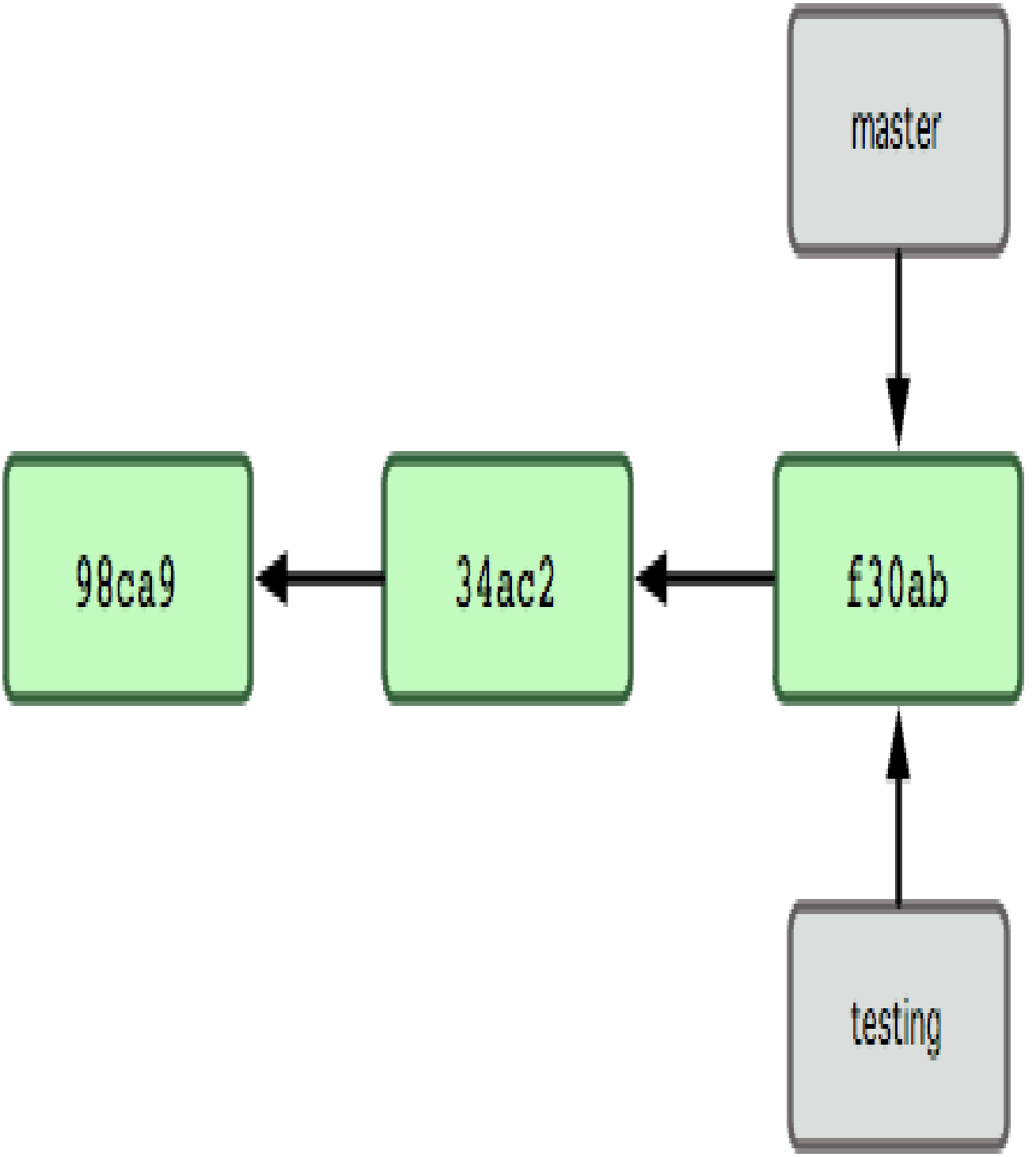


Figura 3-4. Apuntadores de varias ramas en el registro de confirmaciones de cambio.

Y, ¿cómo sabe Git en qué rama estás en este momento? Pues..., mediante un apuntador especial denominado HEAD. Aunque es preciso comentar que este HEAD es totalmente distinto al concepto de HEAD en otros sistemas de control de cambios como Subversion o CVS. En Git, es simplemente el apuntador a la rama local en la que tú estés en ese momento. En este caso, en la rama master. Puesto que el comando git branch solamente crea una nueva rama, y no salta a dicha rama.

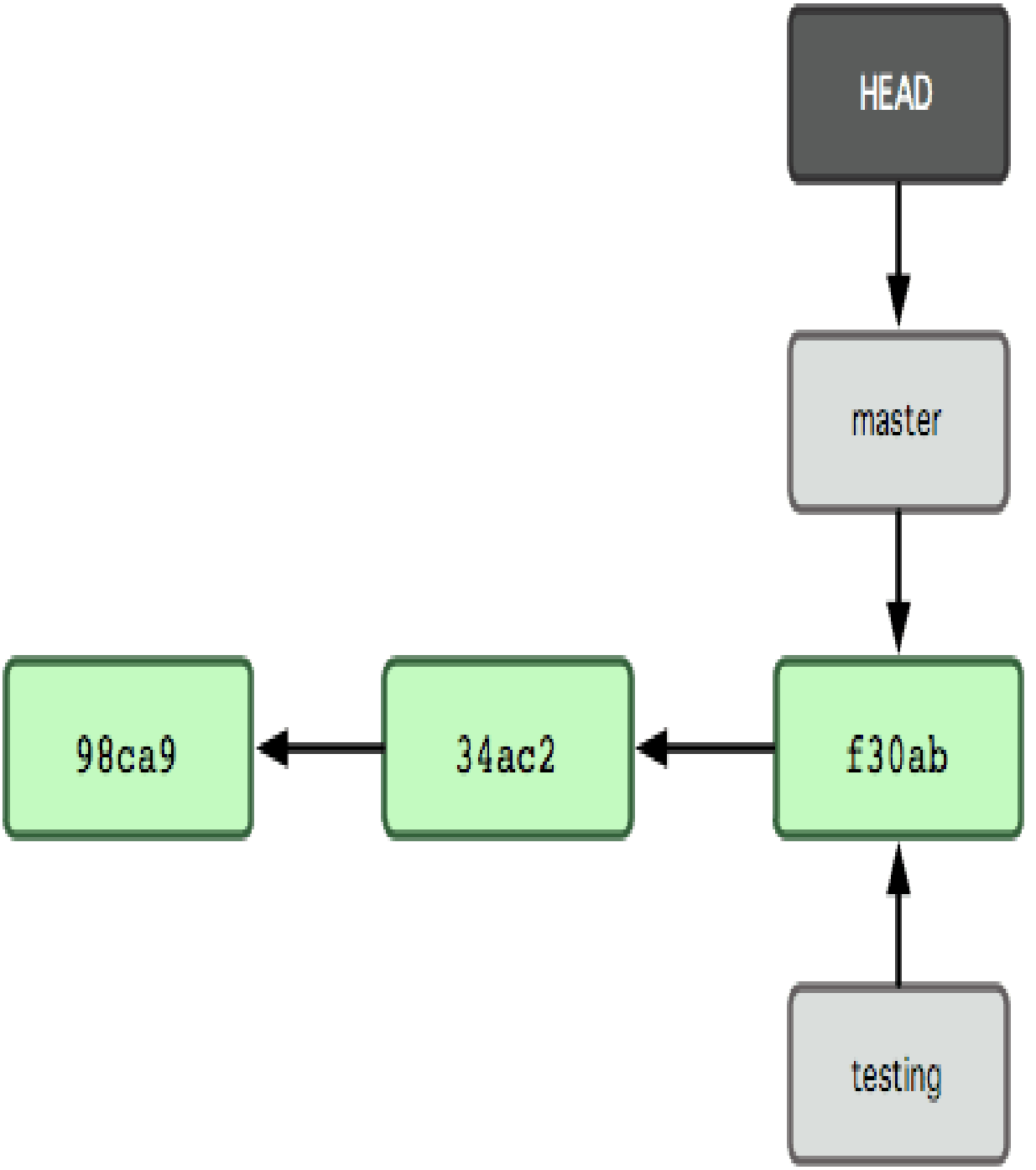


Figura 3-5. Apuntador HEAD a la rama donde estás actualmente.

Para saltar de una rama a otra, tienes que utilizar el comando git checkout. Hagamos una prueba, saltando a la rama testing recién creada:

```
$ git checkout testing
```

Esto mueve el apuntador HEAD a la rama testing (ver Figura 3-6).

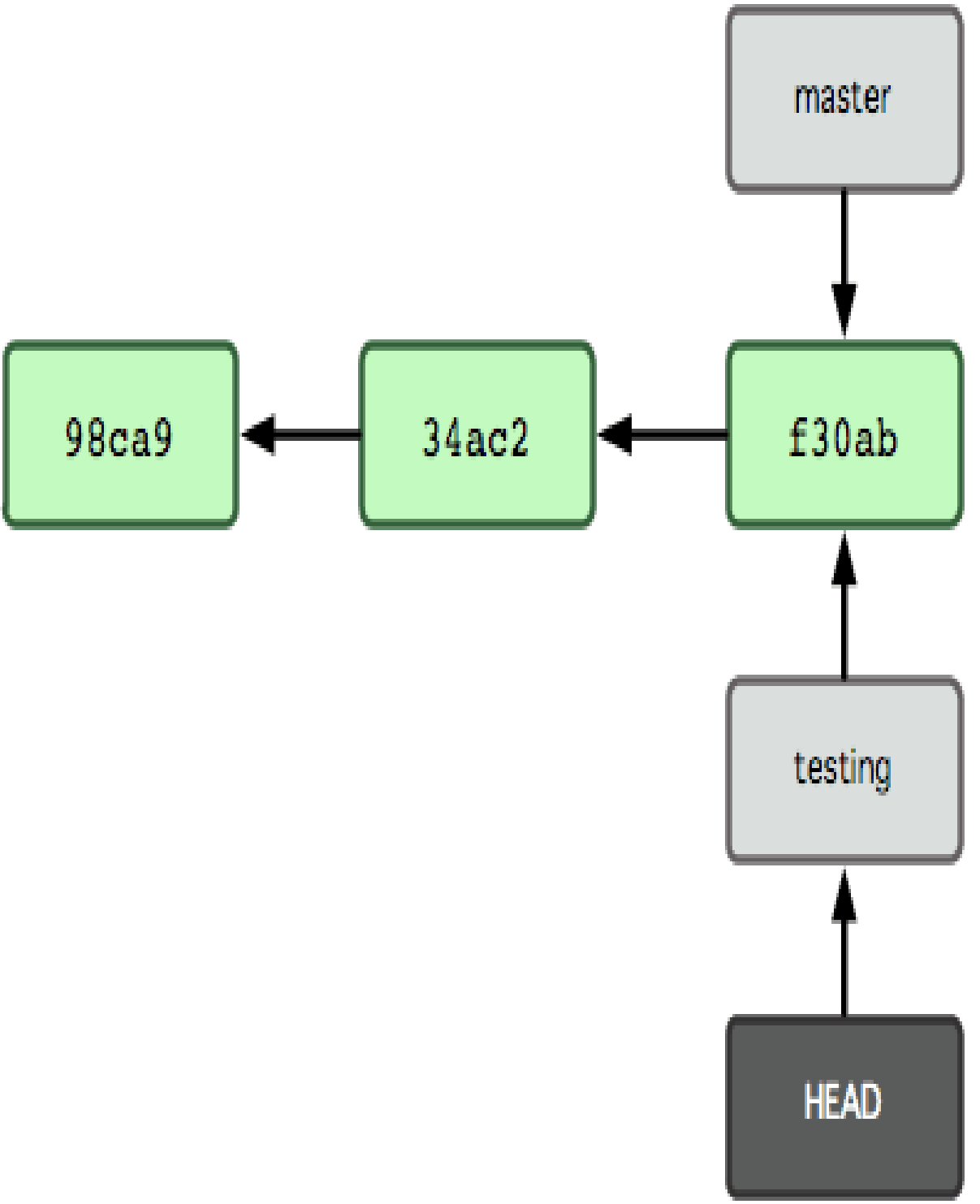


Figura 3-6. Apuntador HEAD apuntando a otra rama cuando saltamos de rama.

¿Cuál es el significado de todo esto?. Bueno... lo veremos tras realizar otra confirmación de cambios:

```
$ vim test.rb
$ git commit -a -m 'made a change'
```

La Figura 3-7 ilustra el resultado.

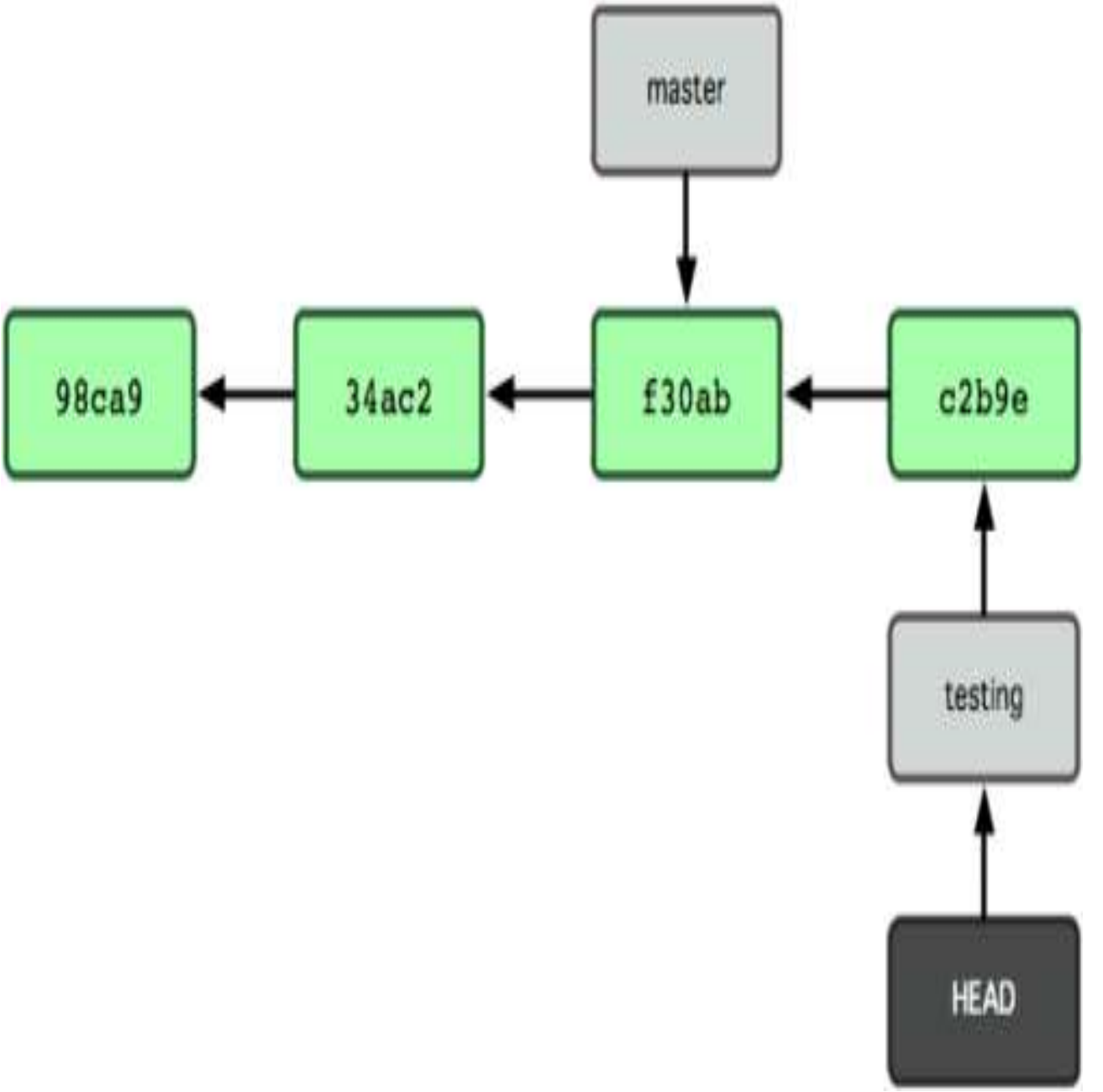


Figura 3-7. La rama apuntada por HEAD avanza con cada confirmación de cambios.

Observamos algo interesante: la rama testing avanza, mientras que la rama master permanece en la confirmación donde estaba cuando lanzaste el comando git checkout para saltar. Volvamos ahora a la rama master:

```
$ git checkout master
```

La Figura 3-8 muestra el resultado.

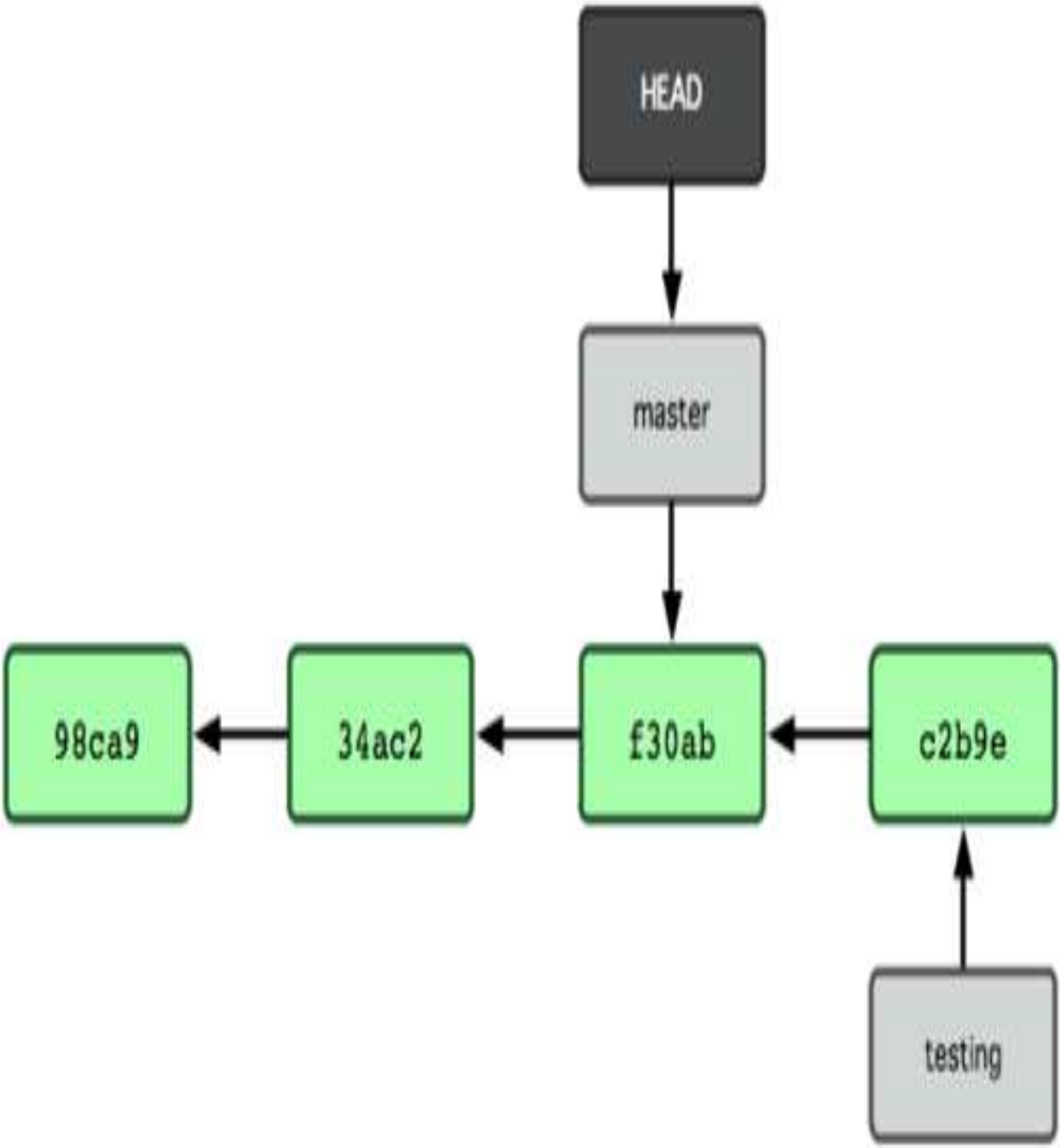


Figura 3-8. HEAD apunta a otra rama cuando hacemos un checkout.

Este comando realiza dos acciones: Mueve el apuntador HEAD de nuevo a la rama master, y revierte los archivos de tu directorio de trabajo; dejándolos tal y como estaban en la última instantánea confirmada en dicha rama master. Esto supone que los cambios que hagas desde este momento en adelante divergirán de la antigua versión del proyecto. Básicamente, lo que se está haciendo es rebobinar el trabajo que habías hecho temporalmente en la rama testing; de tal forma que puedas avanzar en otra dirección diferente.

Haz algunos cambios más y confirmalos:

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

Ahora el registro de tu proyecto diverge (ver Figura 3-9). Has creado una rama y saltado a ella, has trabajado sobre ella; has vuelto a la rama original, y has

trabajado también sobre ella. Los cambios realizados en ambas sesiones de trabajo están aislados en ramas independientes: puedes saltar libremente de una a otra según estimes oportuno. Y todo ello simplemente con dos comandos: `git branch` y `git checkout`.

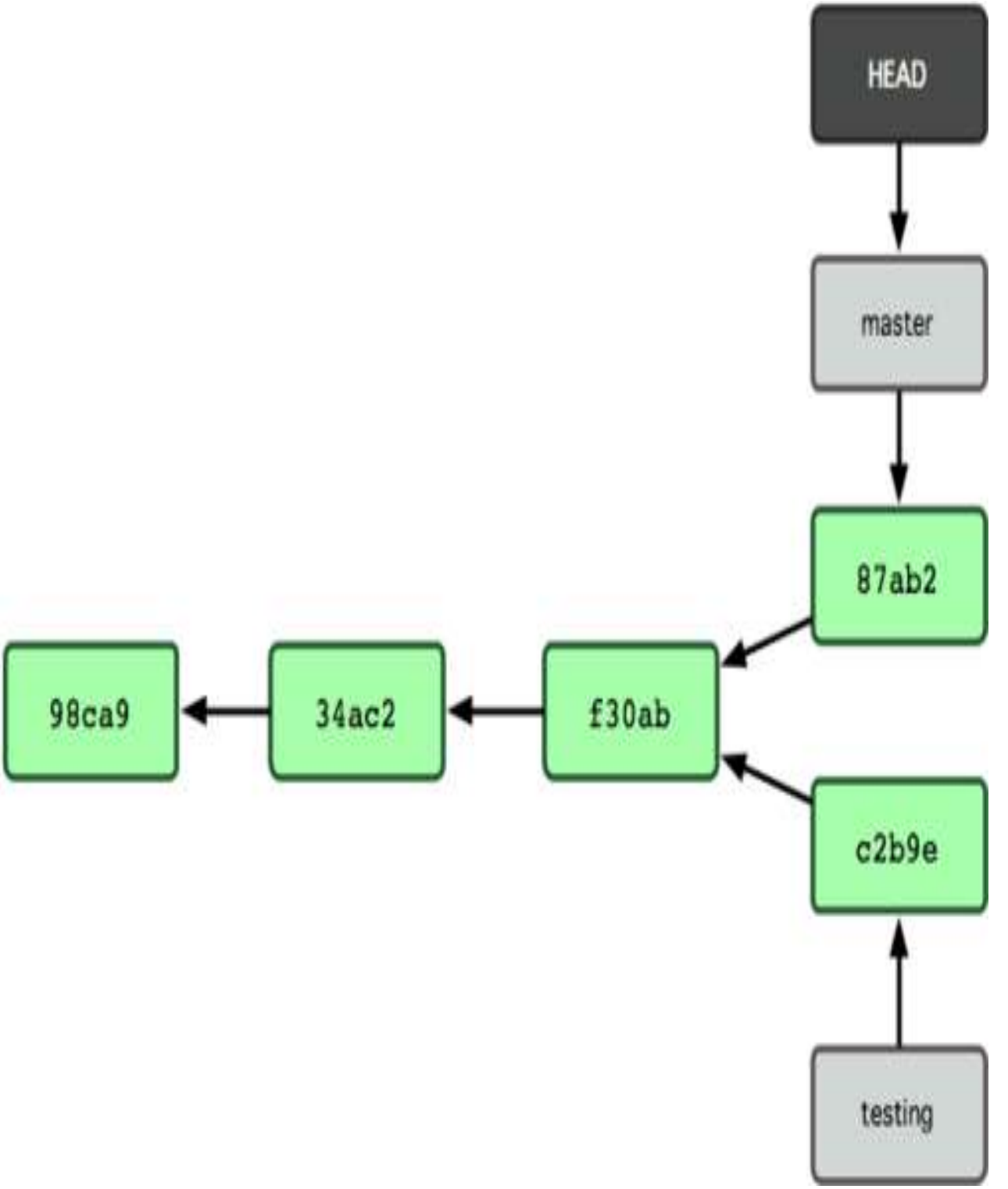


Figura 3-9. Los registros de las ramas divergen.

Debido a que una rama Git es realmente un simple archivo que contiene los 40 caracteres de una suma de control SHA-1, (representando la confirmación de cambios a la que apunta), no cuesta nada el crear y destruir ramas en Git. Crear una nueva rama es tan rápido y simple como escribir 41 bytes en un archivo, (40 caracteres y un retorno de carro).

Esto contrasta fuertemente con los métodos de ramificación usados por otros sistemas de control de versiones. En los que crear una nueva rama supone el copiar todos los archivos del proyecto a una nueva carpeta adicional. Lo que puede llevar segundos o incluso minutos, dependiendo del tamaño del proyecto. Mientras que en Git el proceso es siempre instantáneo. Y, además, debido a que se almacenan también los nodos padre para cada confirmación, el encontrar las bases adecuadas para

realizar una fusión entre ramas es un proceso automático y generalmente sencillo de realizar. Animando así a los desarrolladores a utilizar ramificaciones frecuentemente.

Y vamos a ver el por qué merece la pena hacerlo así.

.2
Procedimientos básicos para ramificar y fusionar

Vamos a presentar un ejemplo simple de ramificar y de fusionar, con un flujo de trabajo que se podría presentar en la realidad. Imagina que sigues los siquientes pasos:

1. **Trabajas en un sitio web.**
2. **Creas una rama para un nuevo tema sobre el que quieres trabajar.**
3. **Realizas algo de trabajo en esa rama.**

En este momento, recibes una llamada avisándote de un problema crítico que has de resolver. Y sigues los siguientes pasos:

1. **Vuelves a la rama de producción original.**
2. **Creas una nueva rama para el problema crítico y lo resuelves trabajando en ella.**
3. **Tras las pertinentes pruebas, fusionas (merge) esa rama y la envías (push) a la rama de producción.**
4. **Vuelves a la rama del tema en que andabas antes de la llamada y continuas tu trabajo.**

Procedimientos básicos de ramificación

Imagina que estas trabajando en un proyecto, y tienes un par de confirmaciones (commit) ya realizadas. (ver Figura 3-10)

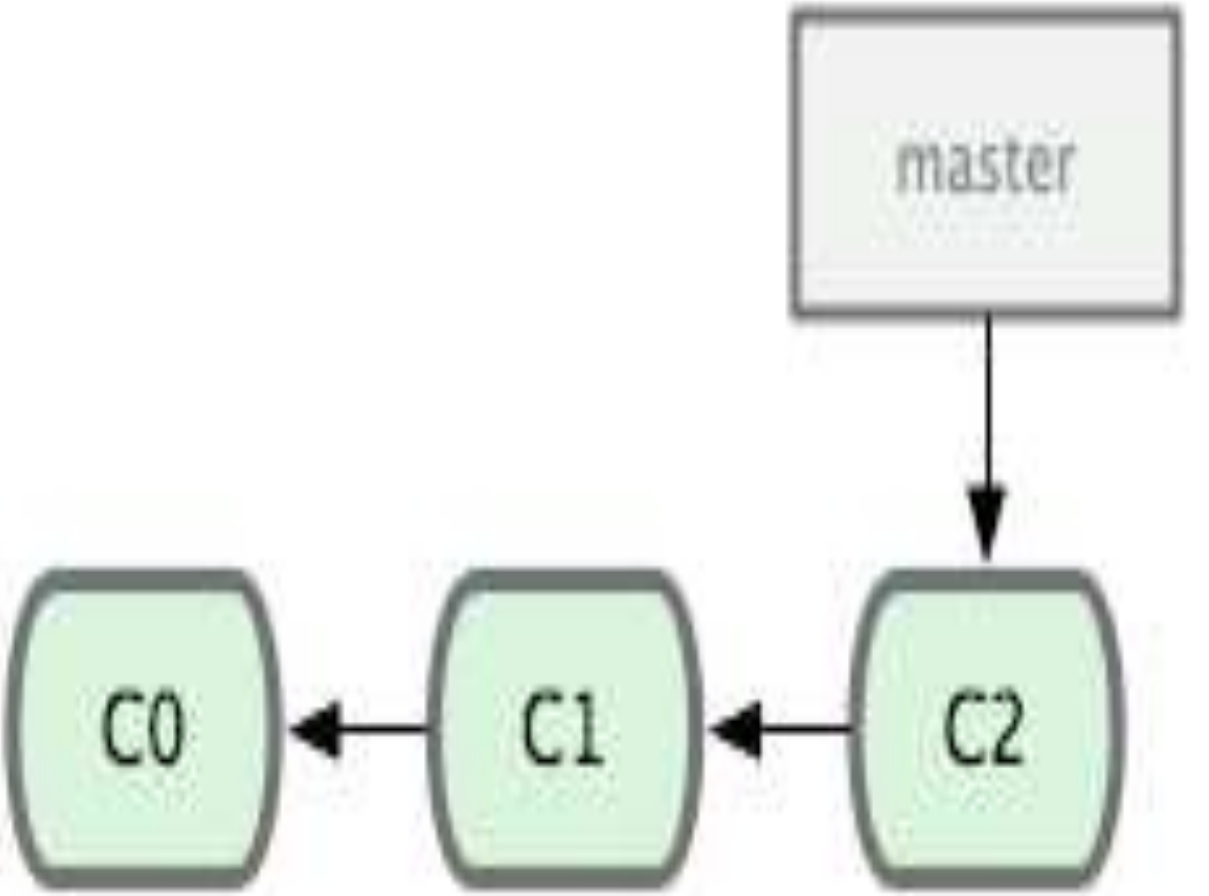


Figura 3-10. Un registro de confirmaciones simple y corto.

Decides trabajar el problema #53, del sistema que tu compañía utiliza para llevar seguimiento de los problemas. Aunque, por supuesto, Git no está ligado a ningún sistema de seguimiento de problemas concreto. Como el problema #53 es un tema concreto y puntual en el que vas a trabajar, creas una nueva rama para él. Para crear una nueva rama y saltar a ella, en un solo paso, puedes utilizar el comando `git checkout` con la opción `-b`:

```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

Esto es un atajo a:

```
$ git branch iss53
$ git checkout iss53
```

Figura 3-11 muestra el resultado.

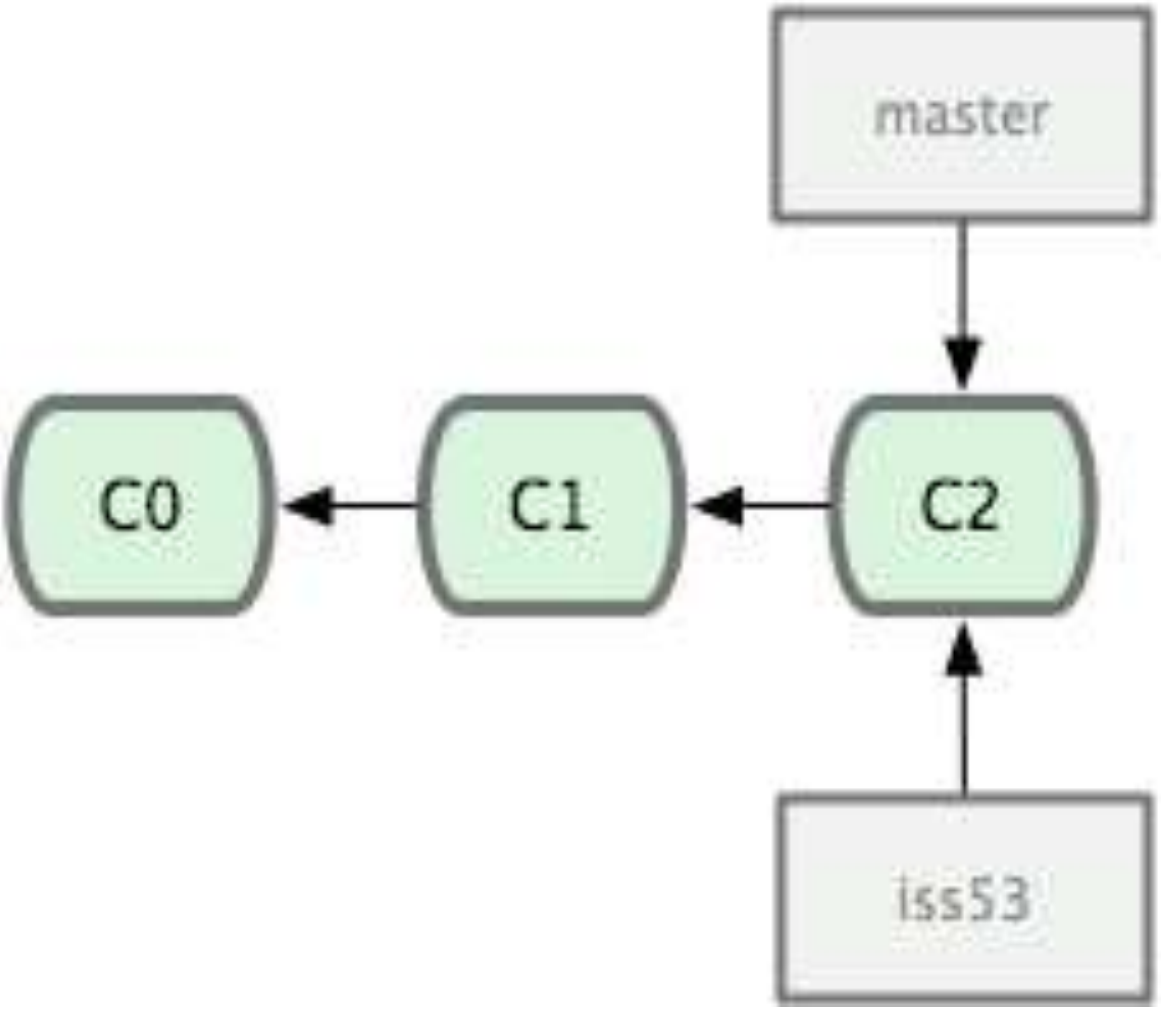


Figura 3-11. Creación de un apuntador a la nueva rama.

Trabajas en el sitio web y haces algunas confirmaciones de cambios (commits). Con ello avanzas la rama iss53, que es la que tienes activada (checked out) en este momento (es decir, a la que apunta HEAD; ver Figura 3-12):

$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'

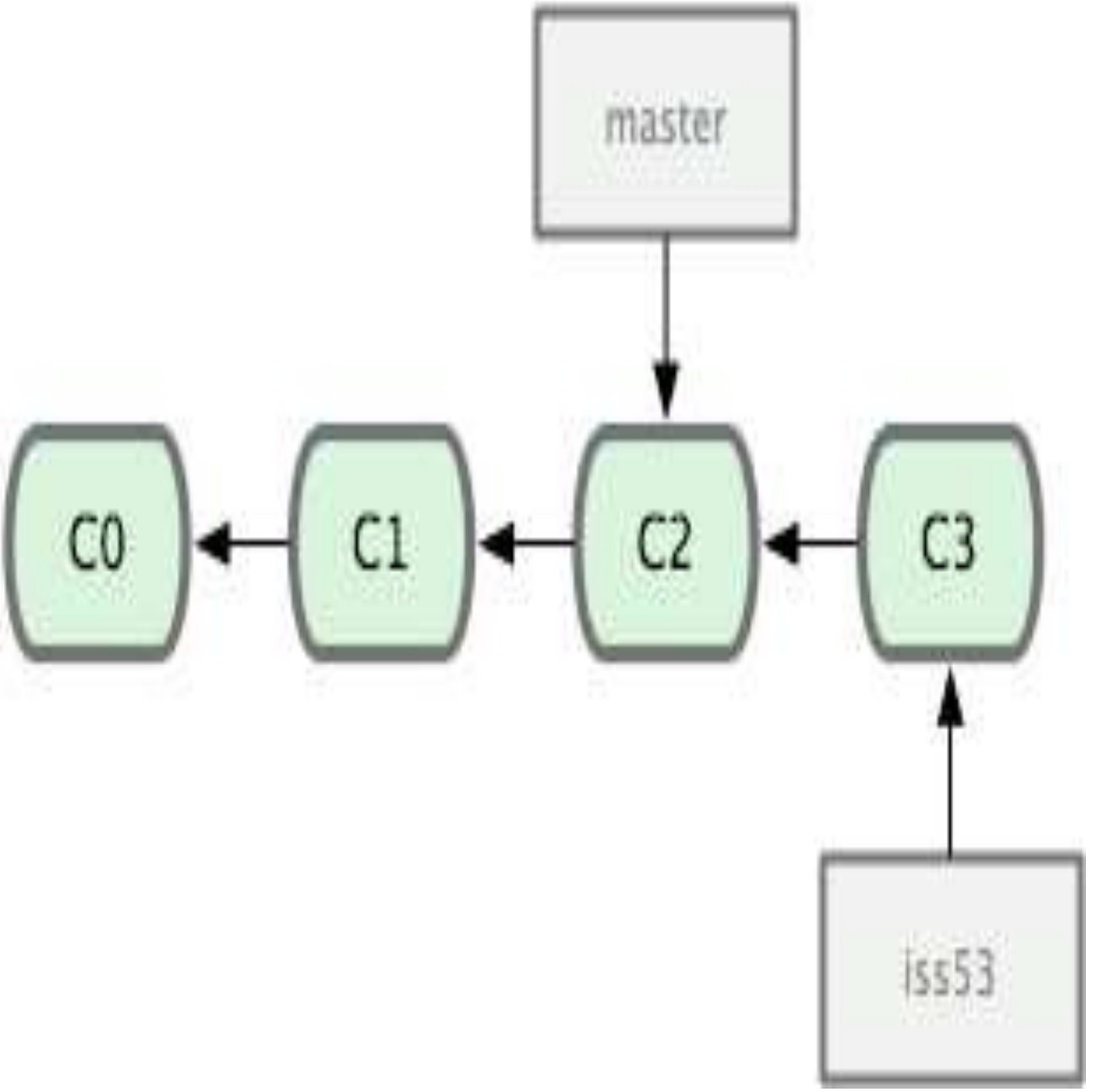


Figura 3-12. La rama iss53 ha avanzado con tu trabajo.

Entonces, recibes una llamada avisándote de otro problema urgente en el sitio web. Problema que has de resolver inmediatamente. Usando Git, no necesitas mezclar el nuevo problema con los cambios que ya habías realizado sobre el problema #53; ni tampoco perder tiempo revirtiendo esos cambios para poder trabajar sobre el contenido que está en producción. Basta con saltar de nuevo a la rama master y continuar trabajando a partir de ella.

Pero, antes de poder hacer eso, hemos de tener en cuenta que teniendo cambios aún no confirmados en la carpeta de trabajo o en el área de preparación, Git no nos permitirá saltar a otra rama con la que podríamos tener conflictos. Lo mejor es tener siempre un estado de trabajo limpio y despejado antes de saltar entre ramas. Y, para ello, tenemos algunos procedimientos (stash y commit ammend), que vamos a ver

más adelante. Por ahora, como tenemos confirmados todos los cambios, podemos saltar a la rama master sin problemas:

```
$ git checkout master
Switched to branch "master"
```

Tras esto, tendrás la carpeta de trabajo exactamente igual a como estaba antes de comenzar a trabajar sobre el problema #53. Y podrás concentrarte en el nuevo problema urgente. Es importante recordar que Git revierte la carpeta de trabajo exactamente al estado en que estaba en la confirmación (commit) apuntada por la rama que activamos (checkout) en cada momento. Git añade, quita y modifica archivos automáticamente. Para asegurarte que tu copia de trabajo es exactamente tal y como era la rama en la última confirmación de cambios realizada sobre ella.

Volviendo al problema urgente. Vamos a crear una nueva rama hotfix, sobre la que trabajar hasta resolverlo (ver Figura 3-13):

```
$ git checkout -b 'hotfix'
Switched to a new branch "hotfix"
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix]: created 3a0874c: "fixed the broken email address"
 1 files changed, 0 insertions(+), 1 deletions(-)
```

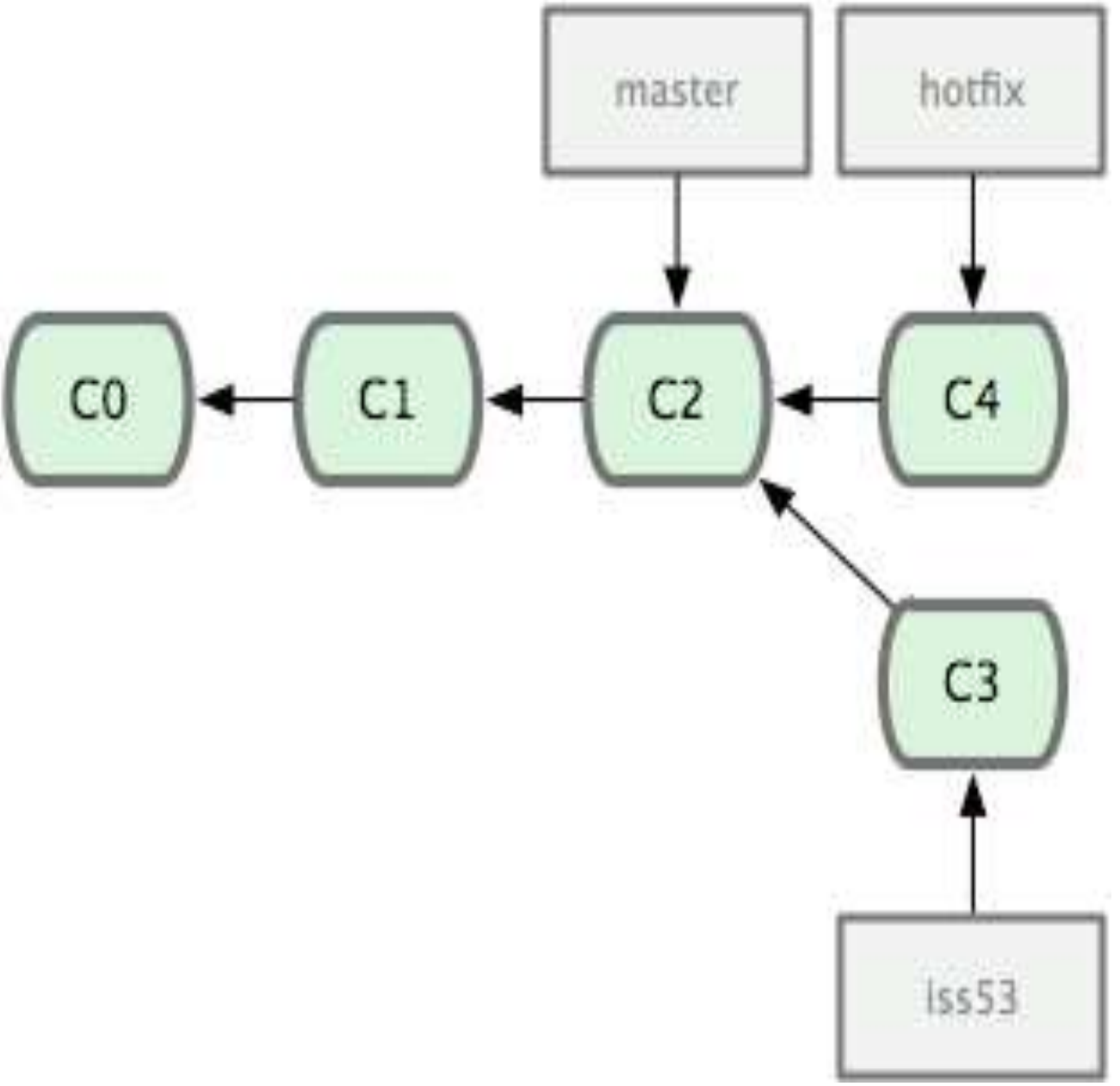


Figura 3-13. rama hotfix basada en la rama master original.

Puedes realizar las pruebas oportunas, asegurarte que la solución es correcta, e incorporar los cambios a la rama master

para ponerlos en producción. Esto se hace con el comando `git merge`:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast forward
 README |    1 -
 1 files changed, 0 insertions(+), 1 deletions(-)
```

Merece destacar la frase "Avance rápido" ("Fast forward") que aparece en la respuesta al comando. Git ha movido el apuntador hacia adelante, ya que la confirmación apuntada en la rama donde has fusionado estaba directamente "aguas arriba" respecto de la confirmación actual. Dicho de otro modo: cuando intentas fusionar una confirmación con otra confirmación accesible siguiendo directamente el registro de la primera; Git simplifica las cosas avanzando el puntero, ya que no hay ningún otro trabajo divergente a

fusionar. Esto es lo que se denomina "avance rápido" ("fast forward").

Ahora, los cambios realizados están ya en la instantánea (snapshot) de la confirmación (commit) apuntada por la rama master. Y puedes desplegarlos (ver Figura 3-14)

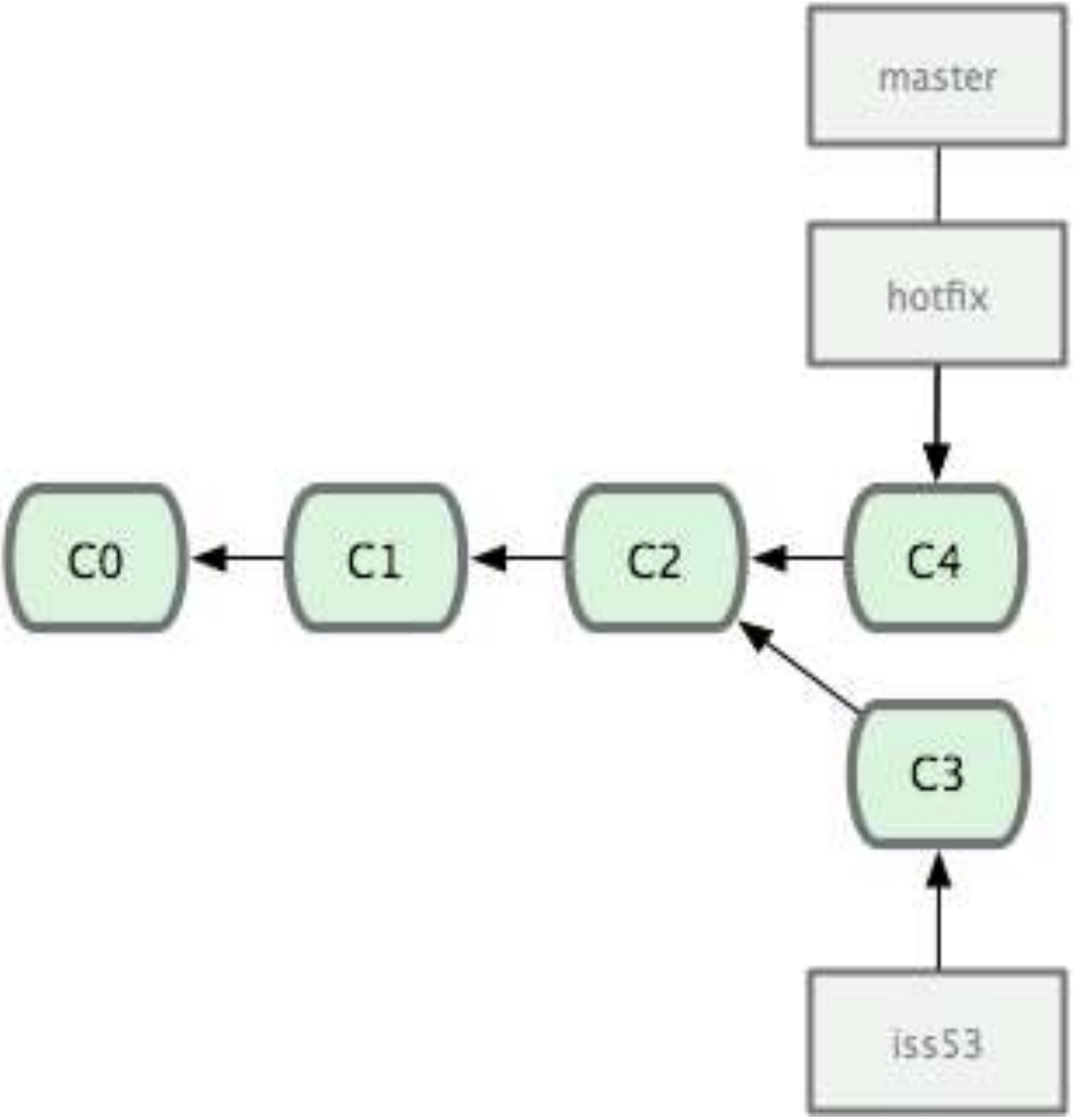


Figura 3-14. Tras la fusión (merge), la rama `master` apunta al mismo sitio que la rama `hotfix`.

Tras haber resuelto el problema urgente que te había interrumpido tu trabajo, puedes volver a donde estabas. Pero antes, es interesante borrar la rama hotfix. Ya que no la vamos a necesitar más, puesto que apunta exactamente al mismo sitio que la rama master. Esto lo puedes hacer con la opción -d del comando git branch:

```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

Y, con esto, ya estás dispuesto para regresar al trabajo sobre el problema #53 (ver Figura 3-15):

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53]: created ad82d7a: "finished the new footer [issue 53]"
 1 files changed, 1 insertions(+), 0 deletions(-)
```

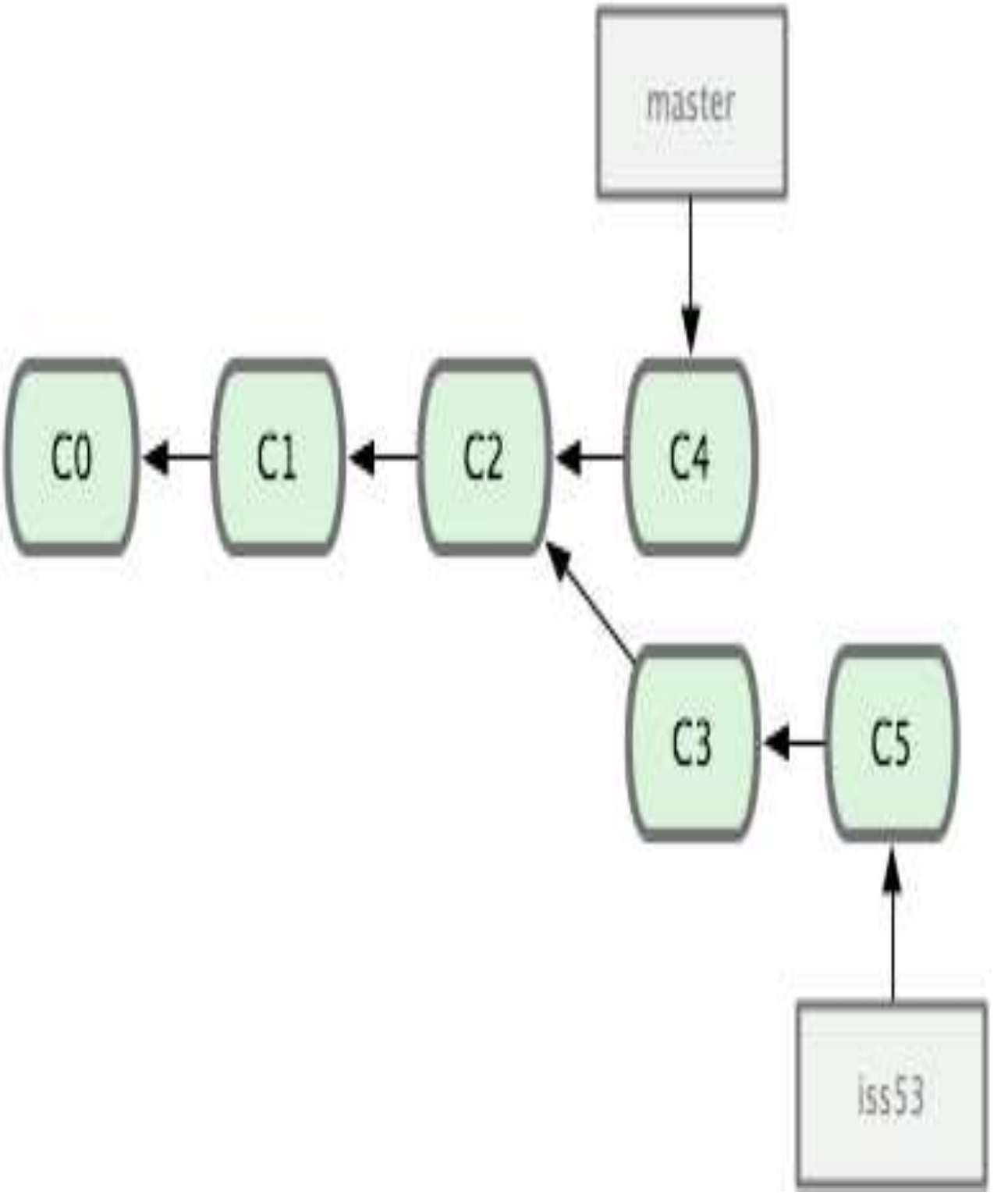


Figura 3-15. La rama iss53 puede avanzar independientemente.

Cabe indicar que todo el trabajo realizado en la rama hotfix no está en los archivos de la rama iss53. Si fuera necesario agregarlos, puedes fusionar (merge) la rama master sobre la rama iss53 utilizando el comando git merge master. O puedes esperar hasta que decidas llevar (pull) la rama iss53 a la rama master.

Procedimientos básicos de fusión

Supongamos que tu trabajo con el problema #53 está ya completo y listo para fusionarlo (merge) con la rama master. Para ello, de forma similar a como antes has hecho con la rama hotfix, vas a fusionar la rama iss53. Simplemente, activando (checkout) la rama donde deseas fusionar y lanzando el comando git merge:

```
$ git checkout master
$ git merge iss53
Merge made by recursive.
 README |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Es algo diferente de la fusión realizada anteriormente con hotfix. En este caso, el registro de desarrollo había divergido en un punto anterior. Debido a que la confirmación en la rama actual no es ancestro directo de la rama que pretendes fusionar, Git tiene cierto

trabajo extra que hacer. Git realizará una fusión a tres bandas, utilizando las dos instantáneas apuntadas por el extremo de cada una de las ramas y por el ancestro común a ambas dos. La figura 3-16 ilustra las tres instantáneas que Git utiliza para realizar la fusión en este caso.

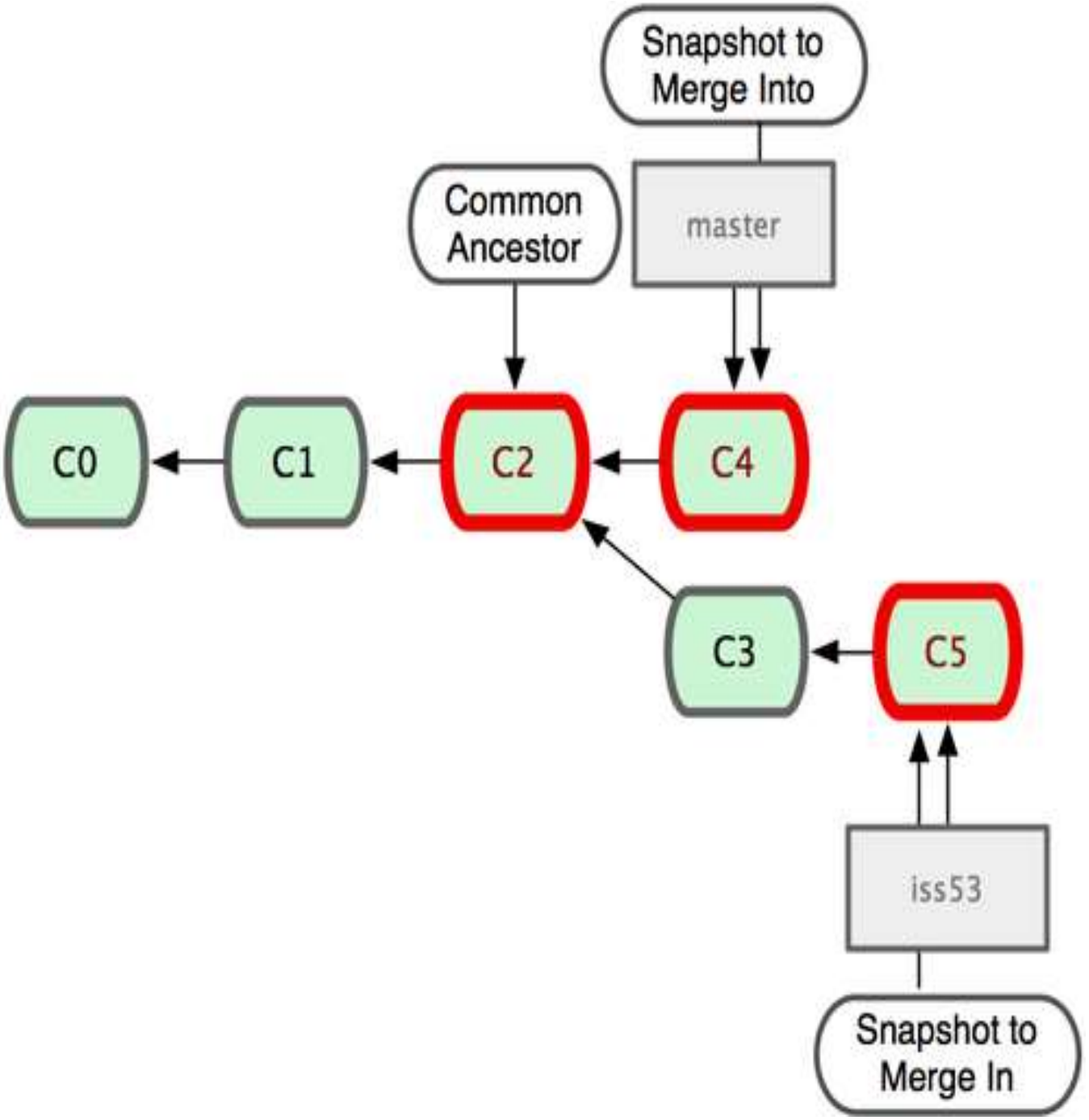


Figura 3-16. Git identifica automáticamente el mejor ancestro común para realizar la fusión de las ramas.

En lugar de simplemente avanzar el apuntador de la rama, Git crea una nueva instantánea (snapshot) resultante de la fusión a tres bandas; y crea automáticamente una nueva confirmación de cambios (commit) que apunta a ella. Nos referimos a este proceso como "fusión confirmada". Y se diferencia en que tiene más de un padre.

Merece la pena destacar el hecho de que es el propio Git quien determina automáticamente el mejor ancestro común para realizar la fusión. Diferenciandose de otros sistemas tales como CVS o Subversion, donde es el desarrollador quien ha de imaginarse cuál puede ser dicho mejor ancestro común. Esto hace que en Git sea mucho más fácil el realizar fusiones.

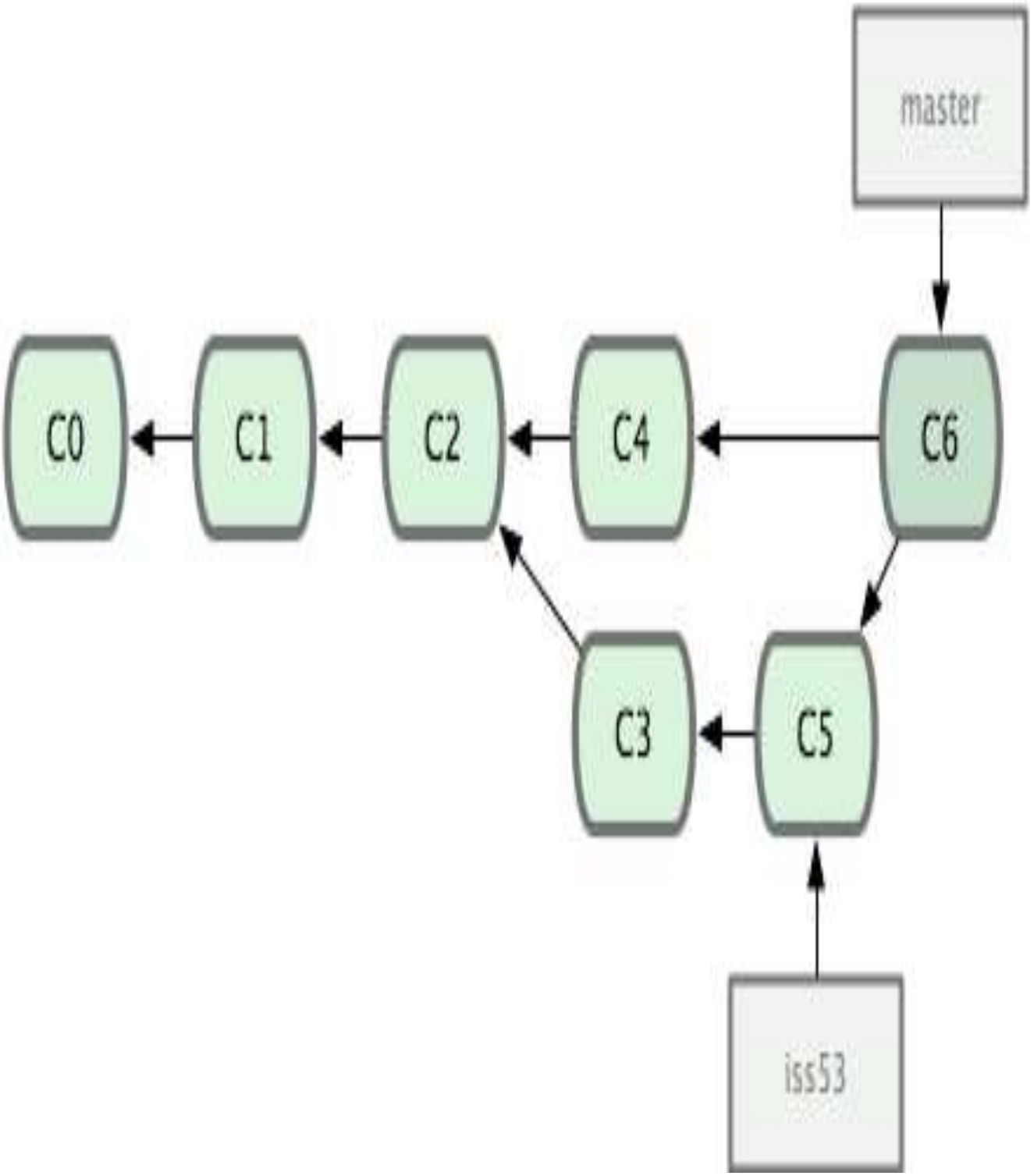


Figura 3-17. Git crea automáticamente una nueva confirmación para la fusión.

Ahora que todo tu trabajo está ya fusionado con la rama principal, ya no tienes necesidad de la rama iss53. Por lo que puedes borrarla. Y cerrar manualmente el problema en el sistema de seguimiento de problemas de tu empresa.

$ git branch -d iss53

Principales conflictos que pueden surgir en las fusiones

En algunas ocasiones, los procesos de fusión no suelen ser fluidos. Si hay modificaciones dispares en una misma porción de un mismo archivo en las dos ramas distintas que pretendes fusionar, Git no será capaz de fusionarlas directamente. Por ejemplo, si en tu trabajo del problema #53 has modificado una misma porción que también ha sido modificada en el problema hotfix. Puedes obtener un conflicto de fusión tal que:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then
commit the result.
```

Git no crea automáticamente una nueva fusión confirmada (merge commit). Sino que hace una pausa en el proceso, esperando a que tu resuelvas el

conflicto. Para ver qué archivos permanecen sin fusionar en un determinado momento conflictivo de una fusión, puedes usar el comando git status:

```
[master*]$ git status
index.html: needs merge
# 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)
#
#   unmerged:   index.html
#
```

Todo aquello que sea conflictivo y no se haya podido resolver, se marca como "sin fusionar" (unmerged). Git añade a los archivos conflictivos unos marcadores especiales de resolución de conflictos. Marcadores que te guiarán cuando abras manualmente los archivos implicados y los edites para corregirlos.

El archivo conflictivo contendrá algo como:

```
<<<<<<< HEAD:index.html
<div id="footer">contact :
email.support@github.com</div>
=======
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

Donde nos dice que la versión en HEAD (la rama master, la que habias activado antes de lanzar el comando de fusión), contiene lo indicado en la parte superior del bloque (todo lo que está encima de =======). Y que la versión en iss53 contiene el resto, lo indicado en la parte inferior del bloque. Para resolver el conflicto, has de elegir manualmente contenido de uno o de otro lado. Por ejemplo, puedes optar por cambiar el bloque, dejándolo tal que:

```
<div id="footer">
```

please contact us at email.support@github.com

Esta corrección contiene un poco de ambas partes. Y se han eliminado completamente las líneas <<<<<<< , ======= y >>>>>>> Tras resolver todos los bloques conflictivos, has de lanzar comandos `git add` para marcar cada archivo modificado. Marcar archivos como preparados (staging), indica a Git que sus conflictos han sido resueltos. Si en lugar de resolver directamente, prefieres utilizar una herramienta gráfica, puedes usar el comando `git mergetool`. Esto arrancará la correspondiente herramienta de visualización y te permitirá ir resolviendo conflictos con ella.

```
$ git mergetool
merge tool candidates: kdiff3 tkdiff xxdiff meld
gvimdiff opendiff emerge vimdiff
Merging the files: index.html

Normal merge conflict for 'index.html':
```

{local}: modified
{remote}: modified
Hit return to start merge resolution tool (opendiff):

Si deseas usar una herramienta distinta de la escogida por defecto (en mi caso opendiff, porque estoy lanzando el comando en un Mac), puedes escogerla entre la lista de herramientas soportadas mostradas al principio ("merge tool candidates"). Tecleando el nombre de dicha herramienta. En el capítulo 7 se verá cómo cambiar este valor por defecto de tu entorno de trabajo.

Tras salir de la herramienta de fusionado, Git preguntará a ver si hemos resuelto todos los conflictos y la fusión ha sido satisfactoria. Si le indicas que así ha sido, Git marca como preparado (staged) el archivo que acabamos de modificar.

En cualquier momento, puedes lanzar el comando `git status` para ver si ya has resuelto todos los conflictos:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   index.html
#
```

Si todo ha ido correctamente, y ves que todos los archivos conflictivos están marcados como preparados, puedes lanzar el comando `git commit` para terminar de confirmar la fusión. El mensaje de confirmación por defecto será algo parecido a:

```
Merge branch 'iss53'

Conflicts:
  index.html
#
# It looks like you may be committing a MERGE.
# If this is not correct, please remove the file
```

 # .git/MERGE_HEAD
 # and try again.
#

Puedes modificar este mensaje añadiendo detalles sobre cómo has resuelto la fusión, si lo consideras útil para que otros entiendan esta fusión en un futuro. Se trata de indicar porqué has hecho lo que has hecho; a no ser que resulte obvio, claro está.

.3 Gestión de ramificaciones

Ahora que ya has creado, fusionado y borrado algunas ramas, vamos a dar un vistazo a algunas herramientas de gestión muy útiles cuando comienzas a utilizar ramas profusamente.

El comando `git branch` tiene más funciones que las de crear y borrar ramas. Si lo lanzas sin argumentos, obtienes una lista de las ramas presentes en tu proyecto:

```
$ git branch
  iss53
* master
  testing
```

Fijate en el carácter `*` delante de la rama `master`: nos indica la rama activa en este

momento. Si hacemos una confirmación de cambios (commit), esa será la rama que avance. Para ver la última confirmación de cambios en cada rama, puedes usar el comando `git branch -v`:

```
$ git branch -v
  iss53   93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

Otra opción útil para averiguar el estado de las ramas, es filtrarlas y mostrar solo aquellas que han sido fusionadas (o que no lo han sido) con la rama actualmente activa. Para ello, Git dispone, desde la versión 1.5.6, las opciones `--merged` y `--no-merged`. Si deseas ver las ramas que han sido fusionadas en la rama activa, puedes lanzar el comando `git branch --merged`:

```
$ git branch --merged
  iss53
* master
```

Aparece la rama iss53 porque ya ha sido fusionada. Y no lleva por delante el caracter * porque todo su contenido ya ha sido incorporado a otras ramas. Podemos borrarla tranquilamente con git branch -d, sin miedo a perder nada.

Para mostrar todas las ramas que contienen trabajos sin fusionar aún, puedes utilizar el comando git branch --no-merged:

```
$ git branch --no-merged
  testing
```

Esto nos muestra la otra rama en el proyecto. Debido a que contiene trabajos sin fusionar aún, al intentarla borrar con git branch -d, el comando nos dará un error:

```
$ git branch -d testing
error: The branch 'testing' is not an ancestor of
your current HEAD.
```

If you are sure you want to delete it, run 'git branch -D testing'.

Si realmente deseas borrar la rama, y perder el trabajo contenido en ella, puedes forzar el borrado con la opción -D; tal y como lo indica el mensaje de ayuda.

4Flujos de trabajo ramificados

Ahora que ya has visto los procedimientos básicos de ramificación y fusión, ¿qué puedes o qué debes hacer con ellos? En este apartado vamos a ver algunos de los flujos de trabajo más comunes, de tal forma que puedas decidir si te gustaría incorporar alguno de ellos a tu ciclo de desarrollo.

Ramas de largo recorrido

Por la sencillez de la fusión a tres bandas de Git, el fusionar de una rama a otra multitud de veces a lo largo del tiempo es fácil de hacer. Esto te posibilita tener varias ramas siempre abiertas, e irlas usando en diferentes etapas del ciclo de desarrollo; realizando frecuentes fusiones entre ellas.

Muchos desarrolladores que usan Git llevan un flujo de trabajo de esta naturaleza, manteniendo en la rama master únicamente el código totalmente estable (el código que ha sido o que va a ser liberado). Teniendo otras ramas paralelas denominadas desarrollo o siguiente, en las que trabajan y realizan pruebas. Estas ramas paralelas no suele estar siempre en un estado estable; pero cada vez que sí lo están, pueden ser fusionadas con la rama master. También es habitual el incorporarle (pull) ramas

puntuales (ramas temporales, como la rama iss53 del anterior ejemplo) cuando las completamos y estamos seguros de que no van a introducir errores.

En realidad, en todo momento estamos hablando simplemente de apuntadores moviendose por la línea temporal de confirmaciones de cambio (commit history). Las ramas estables apuntan hacia posiciones más antiguas en el registro de confirmaciones. Mientras que las ramas avanzadas, las que van abriendo camino, apuntan hacia posiciones más recientes.

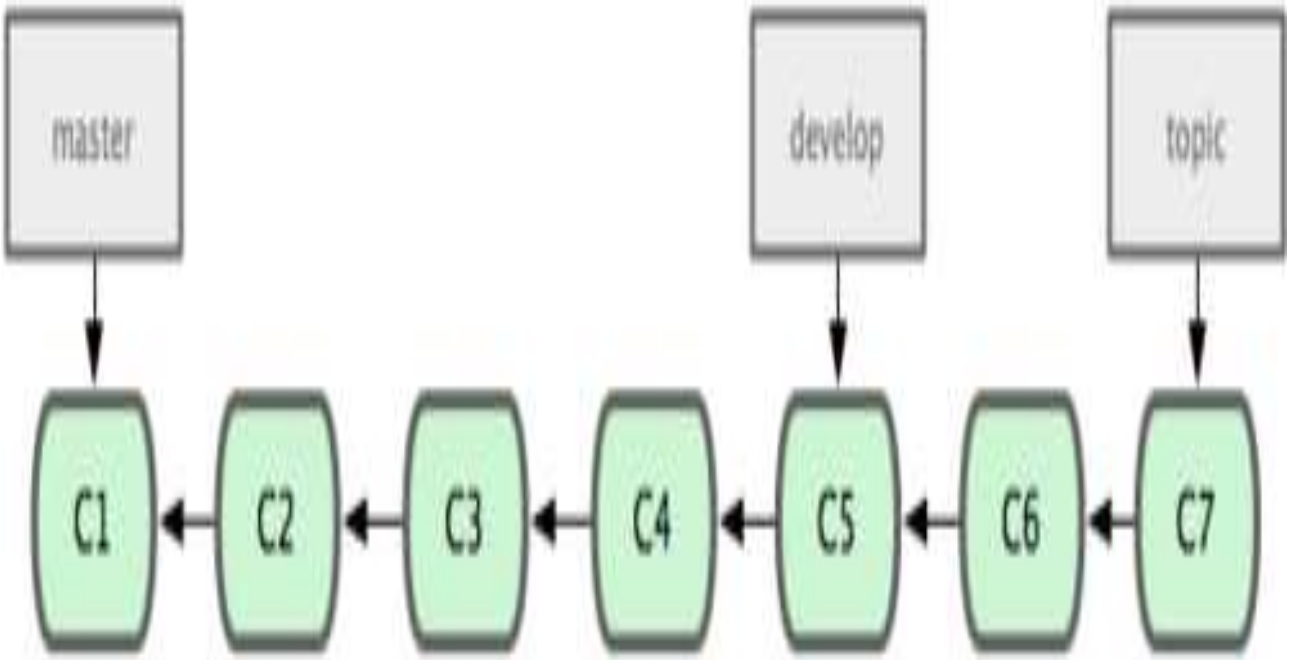


Figura 3-18. Las ramas más estables apuntan hacia posiciones más antiguas en el registro de cambios.

Podría ser más sencillo pensar en las ramas como si fueran silos de almacenamiento. Donde grupos de confirmaciones de cambio (commits) van promocionando hacia silos más estables a medida que son probados y depurados (ver Figura 3-19)

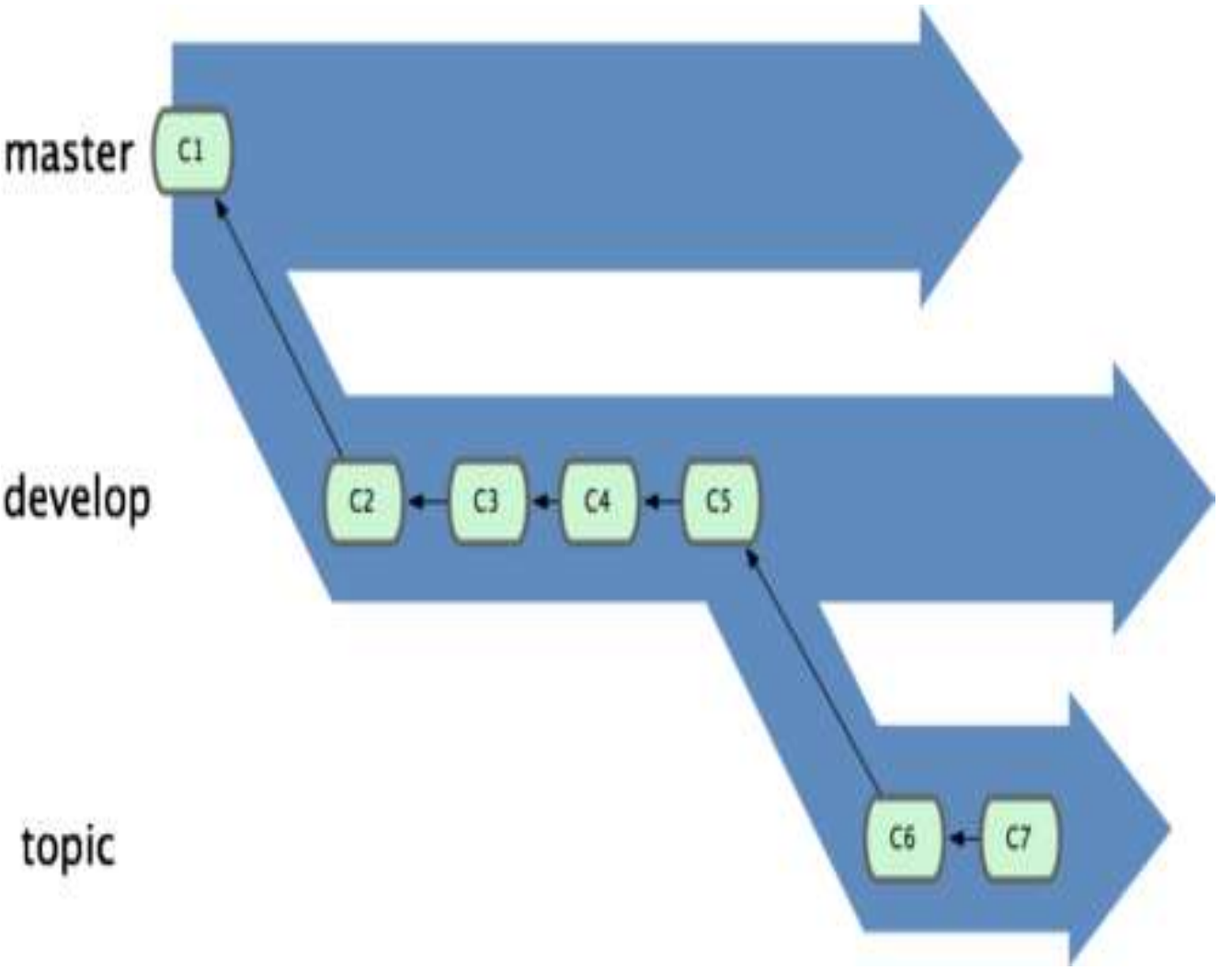


Figura 3-19. Puede ayudar pensar en las ramas como silos de almacenamiento.

Este sistema de trabajo se puede ampliar para diversos grados de estabilidad. Algunos proyectos muy grandes suelen tener una rama denominada propuestas o pu (proposed updates). Donde suele estar todo aquello

integrado desde otras ramas, pero que aún no está listo para ser incorporado a las ramas siguiente o master. La idea es mantener siempre diversas ramas en diversos grados de estabilidad; pero cuando alguna alcanza un estado más estable, la fusionamos con la rama inmediatamente superior a ella. Aunque no es obligatorio el trabajar con ramas de larga duración, realmente es práctico y útil. Sobre todo en proyectos largos o complejos.

Ramas puntuales

Las ramas puntuales, en cambio, son útiles en proyectos de cualquier tamaño. Una rama puntual es aquella de corta duración que abres para un tema o para una funcionalidad muy concretos. Es algo que nunca habrías hecho en otro sistema VCS, debido a los altos costos de crear y fusionar ramas que se suelen dar en esos sistemas. Pero en Git, por el contrario, es muy habitual el crear, trabajar con, fusionar y borrar ramas varias veces al día.

Tal y como has visto con las ramas iss53 y hotfix que has creado en la sección anterior. Has hecho unas pocas confirmaciones de cambio en ellas, y luego las has borrado tras fusionarlas con la rama principal. Esta técnica te posibilita realizar rápidos y completos saltos de contexto. Y, debido a que el trabajo está claramente separado en

silos, con todos los cambios de cada tema en su propia rama, te será mucho más sencillo revisar el código y seguir su evolución. Puedes mantener los cambios ahí durante minutos, dias o meses; y fusionarlos cuando realmente estén listos. En lugar de verte obligado a fusionarlos en el orden en que fueron creados y comenzaste a trabajar en ellos.

Por ejemplo, puedes realizar cierto trabajo en la rama master, ramificar para un problema concreto (rama iss91), trabajar en él un rato, ramificar a una segunda rama para probar otra manera de resolverlo (rama iss92v2), volver a la rama master y trabajar un poco más, y, por último, ramificar temporalmente para probar algo de lo que no estás seguro (rama dumbidea). El registro de confirmaciones (commit history) será algo parecido a la Figura 3-20.

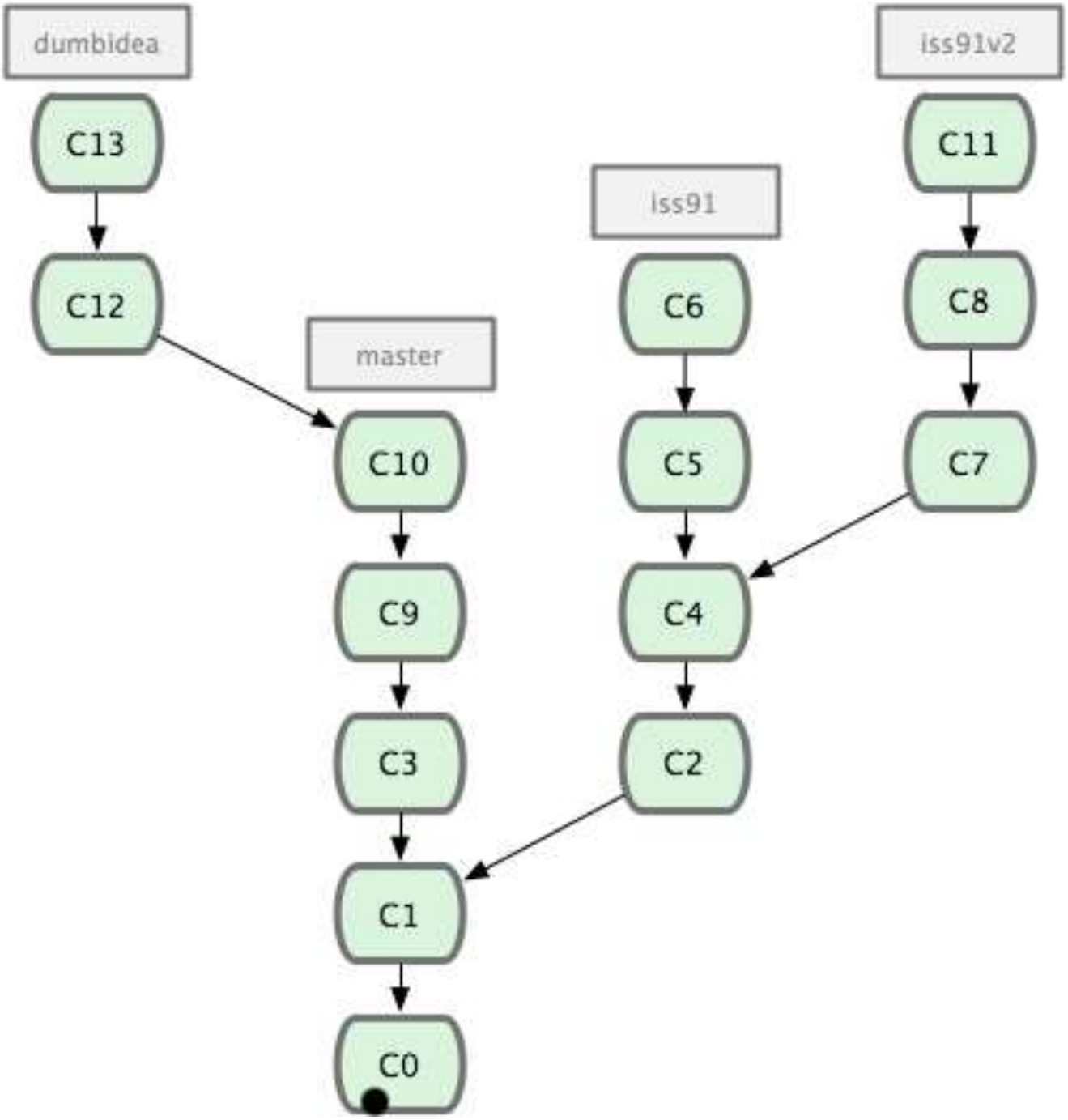


Figura 3-20. El registro de confirmaciones con múltiples ramas puntuales.

En este momento, supongamos que te decides por la segunda solución al problema (rama `iss92v2`); y que, tras mostrar la rama `dumbidea` a tus compañeros, resulta que les parece una idea genial. Puedes descartar la rama `iss91` (perdiendo las confirmaciones C5 y C6), y fusionar las otras dos. El registro será algo parecido a la Figura 3-21.

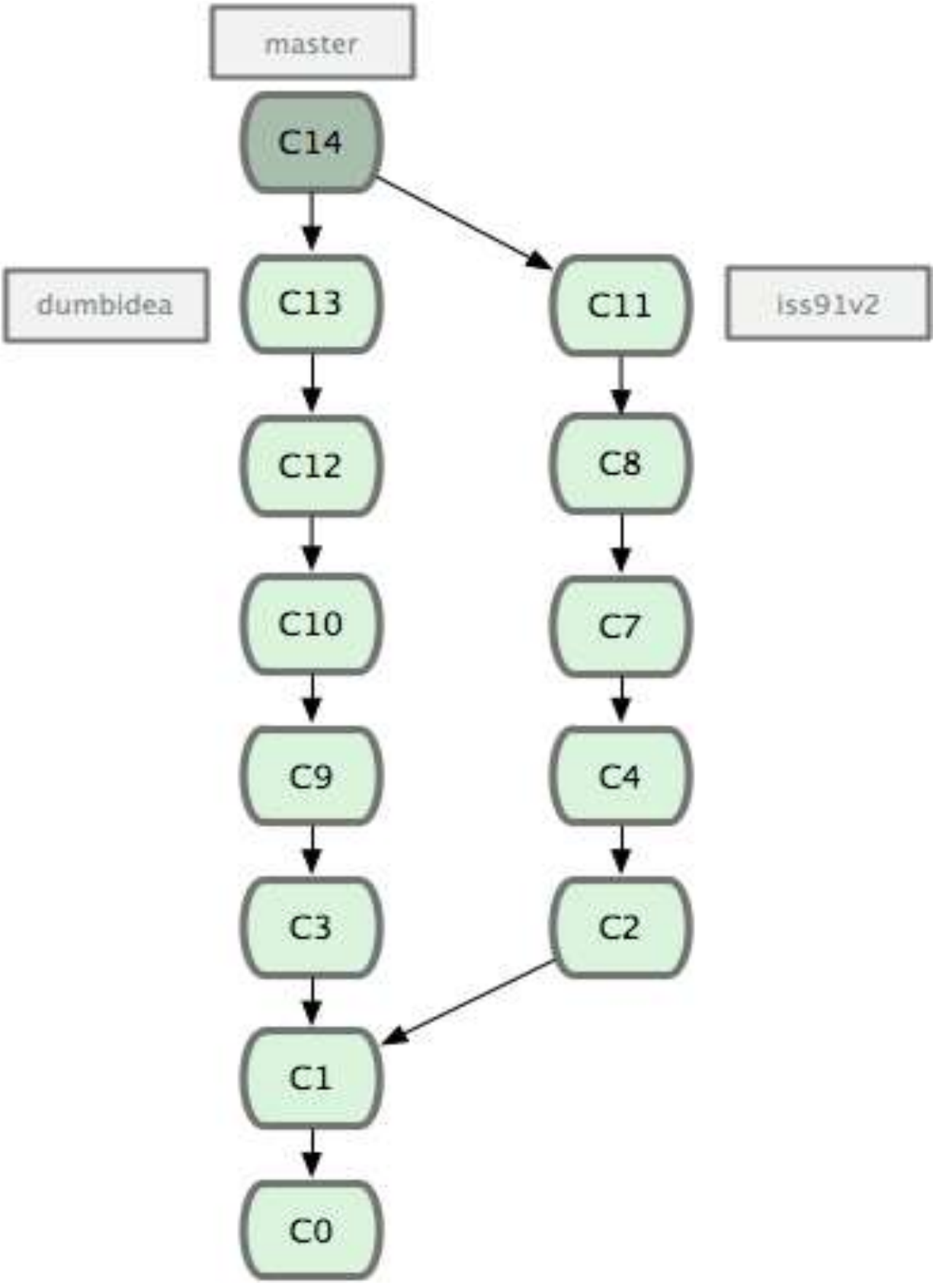


Figura 3-21. El registro tras fusionar dumbidea e iss91v2.

Es importante recordar que, mientras estás haciendo todo esto, todas las ramas son completamente locales. Cuando ramificas y fusionas, todo se realiza en tu propio repositorio Git. No hay nigún tipo de tráfico con ningún servidor.

5 Ramas Remotas

Las ramas remotas son referencias al estado de ramas en tus repositorios remotos. Son ramas locales que no puedes mover; se mueven automáticamente cuando estableces comunicaciones en la red. Las ramas remotas funcionan como marcadores, para recordarte en qué estado se encontraban tus repositorios remotos la última vez que conectaste con ellos.

Suelen referenciarse como (remoto)/(rama). Por ejemplo, si quieres saber cómo estaba la rama master en el remoto origin. Puedes revisar la rama origin/master. O si estás trabajando en un problema con un compañero y este envia (push) una rama iss53, tu tendrás tu propia rama de

trabajo local iss53; pero la rama en el servidor apuntará a la última confirmación (commit) en la rama origin/iss53.

Esto puede ser un tanto confuso, pero intentemos aclararlo con un ejemplo. Supongamos que tienes un sevidor Git en tu red, en git.ourcompany.com. Si haces un clón desde ahí, Git automáticamente lo denominará origin, traerá (pull) sus datos, creará un apuntador hacia donde esté en ese momento su rama master, denominará la copia local origin/master; y será inamovible para ti. Git te proporcionará también tu propia rama master, apuntando al mismo lugar que la rama master de origin; siendo en esta última donde podrás trabajar.

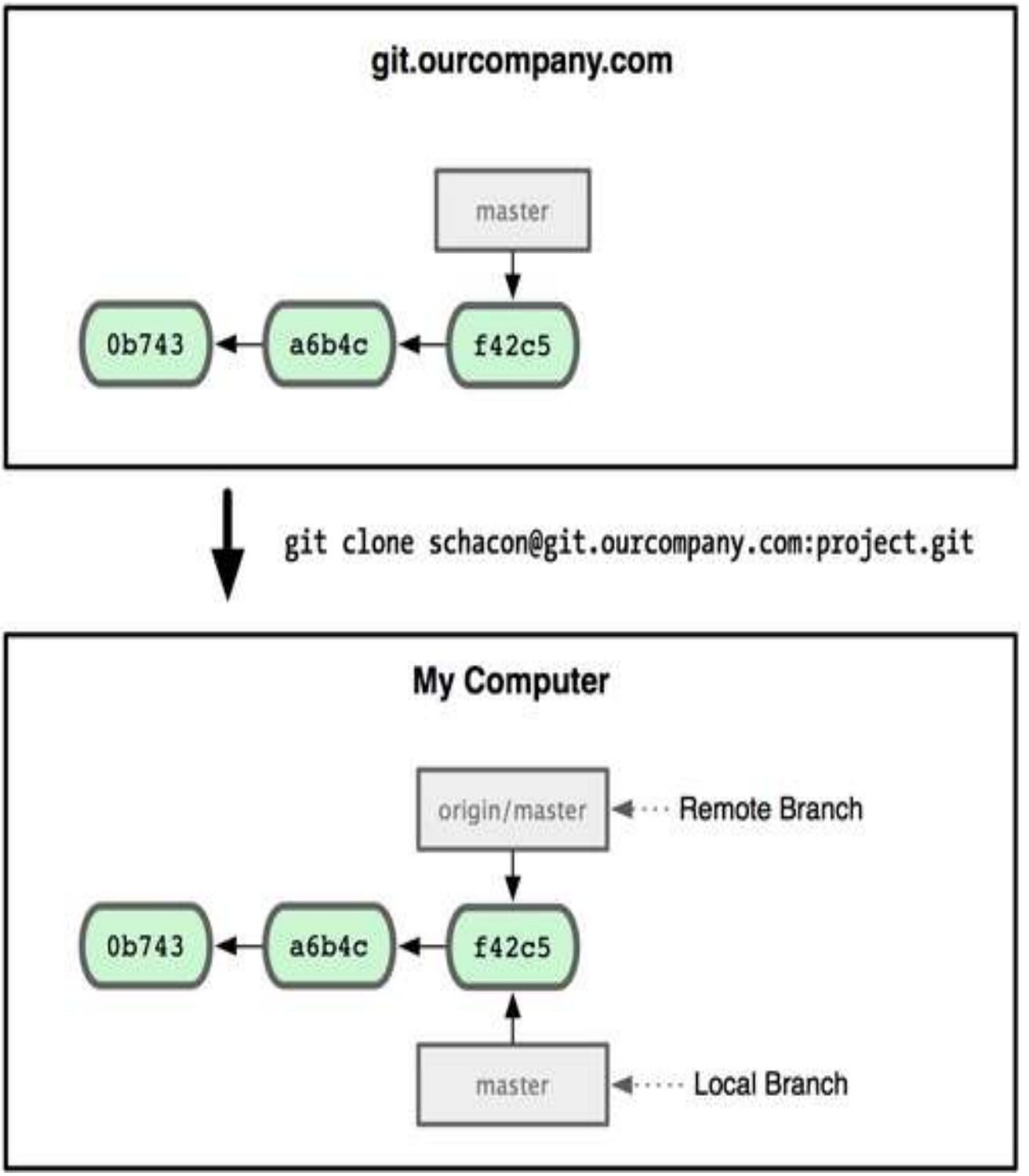


Figura 3-22. Un clón Git te proporciona tu propia rama master y otra rama origin/master apuntando a la rama master original.

Si haces algún trabajo en tu rama `master` local, y al mismo tiempo, alguna otra persona lleva (push) su trabajo al servidor `git.ourcompany.com`, actualizando la rama `master` de allí, te encontrarás con que ambos registros avanzan de forma diferente. Además, mientras no tengas contacto con el servidor, tu apuntador a tu rama `origin/master` no se moverá (ver Figura 3/23).

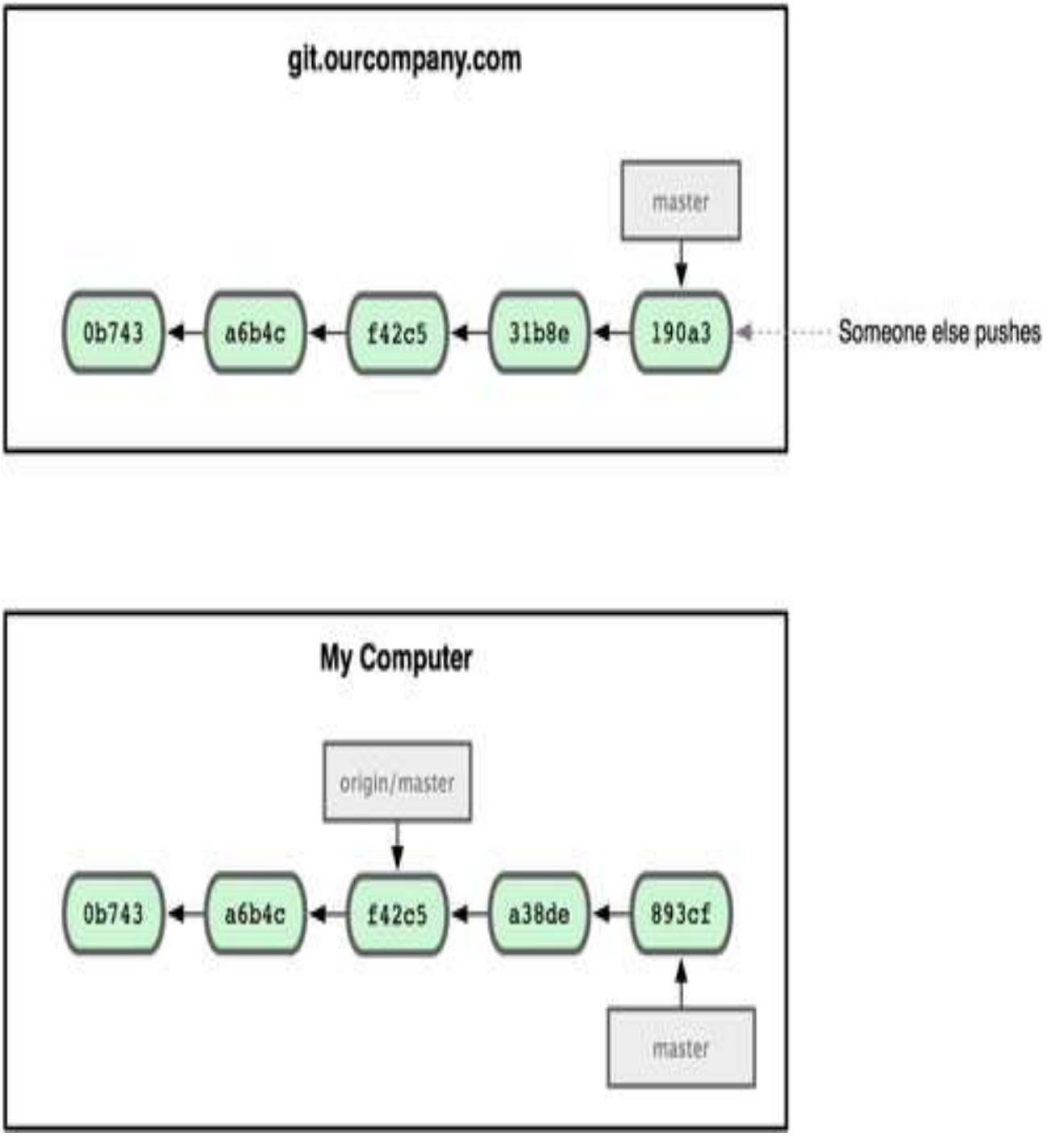


Figura 3-23. Trabajando localmente y que otra persona esté llevando (push) algo al servidor remoto, hace que cada registro avance de forma distinta.

Para sincronizarte, puedes utilizar el comando `git fetch origin`. Este comando localiza en qué servidor está el origen (en este caso `git.ourcompany.com`), recupera cualquier dato presente allí que tu no tengas, y actualiza tu base de datos local, moviendo tu rama `origin/master` para que apunte a esta nueva y más reciente posición (ver Figura 3-24).

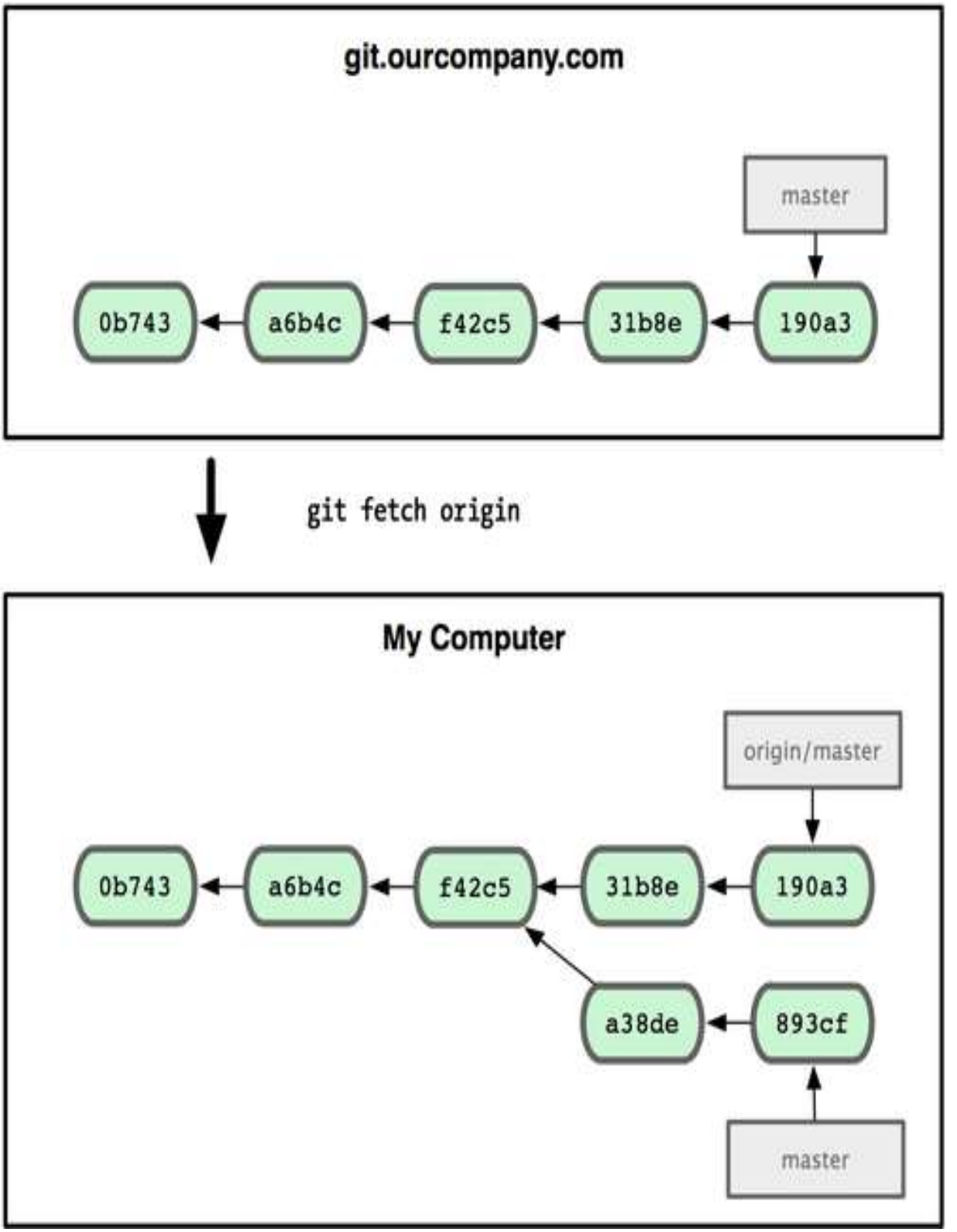


Figura 3-24. El comando git fetch actualiza tus referencias remotas.

Para ilustrar mejor el caso de tener múltiples servidores y cómo van las ramas remotas para esos proyectos remotos. Supongamos que tienes otro servidor Git; utilizado solamente para desarrollo, por uno de tus equipos sprint. Un servidor en `git.team1.ourcompany.com`. Puedes incluirlo como una nueva referencia remota a tu proyecto actual, mediante el comando `git remote add`, tal y como vimos en el capítulo 2. Puedes denominar `teamone` a este remoto, poniendo este nombre abreviado para la URL (ver Figura 3-25)

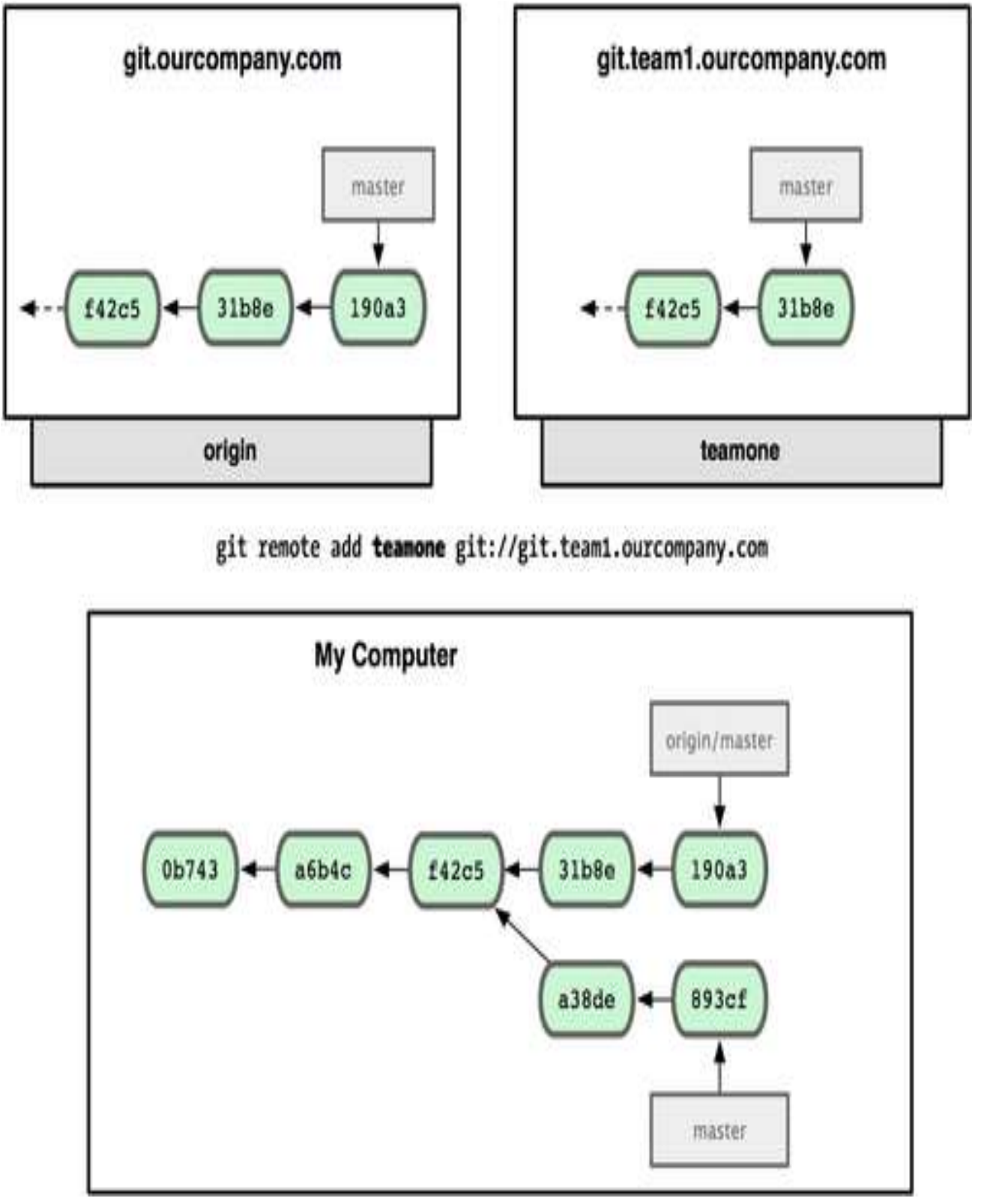


Figura 3-25. Añadiendo otro servidor como remoto.

Ahora, puedes usar el comando `git fetch teamone` para recuperar todo el contenido del servidor que tu no tenias. Debido a que dicho servidor es un subconjunto de de los datos del servidor `origin` que tienes actualmente, Git no recupera (fetch) ningún dato; simplemente prepara una rama remota llamada `teamone/master` para apuntar a la confirmación (commit) que `teamone` tiene en su rama `master`.

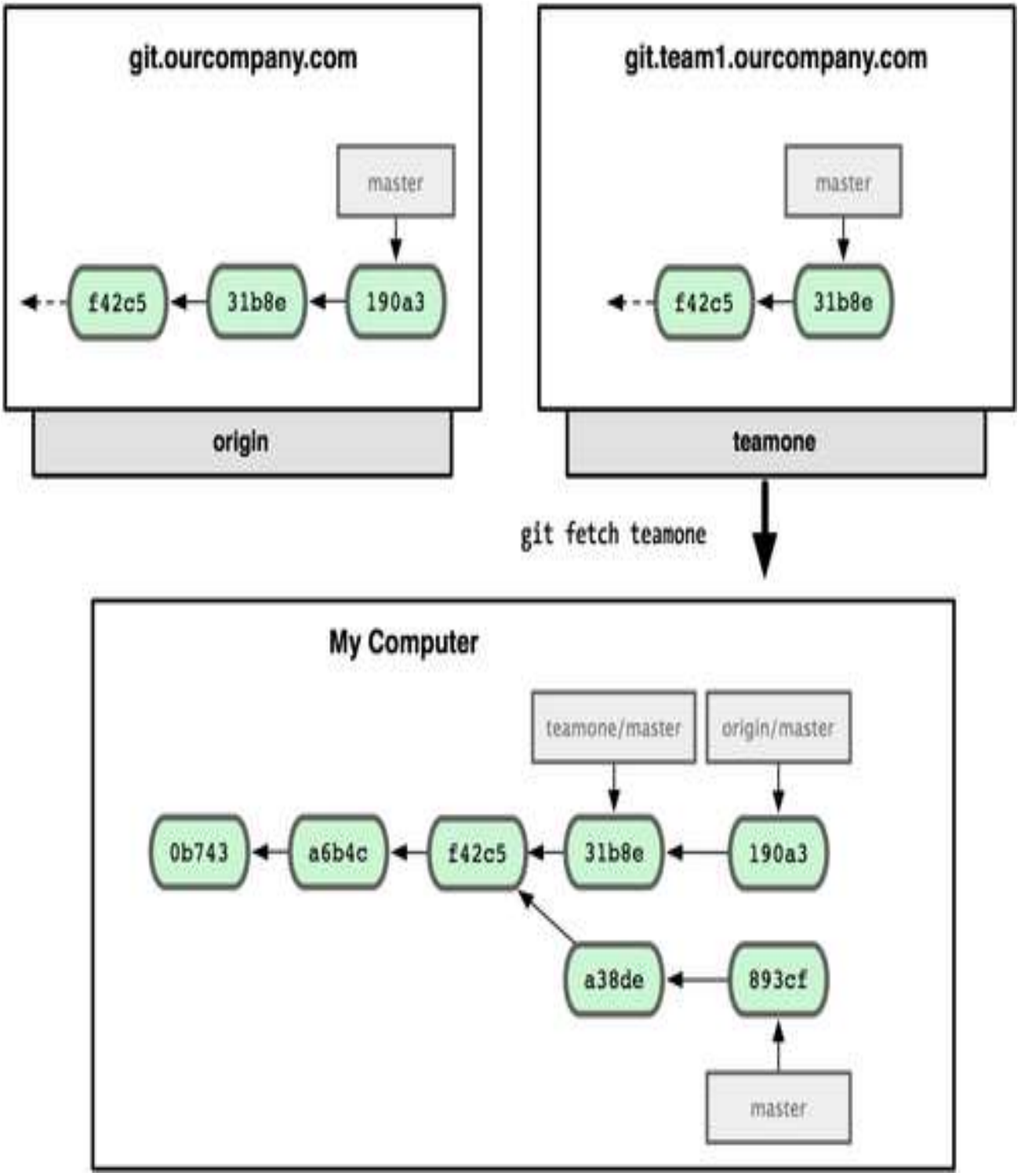


Figura 3-26. Obtienes una referencia local a la posición en la rama master de teamone.

Publicando

Cuando quieres compartir una rama con el resto del mundo, has de llevarla (push) a un remoto donde tengas permisos de escritura. Tus ramas locales no se sincronizan automáticamente con los remotos en los que escribes. Sino que tienes que llevar (push) expresamente, cada vez, al remoto las ramas que desees compartir. De esta forma, puedes usar ramas privadas para el trabajo que no deseas compartir. Llevando a un remoto tan solo aquellas partes que deseas aportar a los demás.

Si tienes una rama llamada serverfix, con la que vas a trabajar en colaboración; puedes llevarla al remoto de la misma forma que llevaste tu primera rama. Con el comando git push (remoto) (rama):

```
$ git push origin serverfix
Counting objects: 20, done.
```

```
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 1.74 KiB, done.
Total 15 (delta 5), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new branch]      serverfix -> serverfix
```

Esto es un poco como un atajo. Git expande automáticamente el nombre de rama `serverfix` a `refs/heads/serverfix:refs/heads/serverfix`, que significa: "coge mi rama local `serverfix` y actualiza con ella la rama `serverfix` del remoto". Volveremos más tarde sobre el tema de `refs/heads/`, viéndolo en detalle en el capítulo 9; aunque puedes ignorarlo por ahora. También puedes hacer `git push origin serverfix:serverfix`, que hace lo mismo; es decir: "coge mi `serverfix` y hazlo el `serverfix` remoto". Puedes utilizar este último formato para llevar una rama local a una rama remota con otro nombre distinto. Si no quieres que se llame `serverfix` en el remoto, puedes lanzar, por ejemplo, `git push origin serverfix:awesomebranch`; para llevar tu

rama serverfix local a la rama awesomebranch en el proyecto remoto.

La próxima vez que tus colaboradores recuperen desde el servidor, obtendrán bajo la rama remota origin/serverfix una referencia a donde esté la versión de serverfix en el servidor:

```
$ git fetch origin
remote: Counting objects: 20, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 15 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (15/15), done.
From git@github.com:schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

Es importante destacar que cuando recuperas (fetch) nuevas ramas remotas, no obtienes automáticamente una copia editable local de las mismas. En otras palabras, en este caso, no tienes una nueva rama serverfix. Sino que únicamente tienes un puntero no editable a origin/serverfix.

Para integrar (merge) esto en tu actual rama de trabajo, puedes usar el comando git merge origin/serverfix. Y si quieres tener tu propia rama serverfix, donde puedas trabajar, puedes crearla directamente basaádote en rama remota:

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch
refs/remotes/origin/serverfix.
Switched to a new branch "serverfix"Switched to
a new branch "serverfix"
```

Esto sí te da una rama local donde puedes trabajar, comenzando donde origin/serverfix estaba en ese momento.

Haciendo seguimiento a las ramas

Activando (checkout) una rama local a partir de una rama remota, se crea automáticamente lo que podríamos denominar "una rama de seguimiento" (tracking branch). Las ramas de seguimiento son ramas locales que tienen una relación directa con alguna rama remota. Si estás en una rama de seguimiento y tecleas el comando `git push`, Git sabe automáticamente a qué servidor y a qué rama ha de llevar los contenidos. Igualmente, tecleando `git pull` mientras estamos en una de esas ramas, recupera (fetch) todas las referencias remotas y las consolida (merge) automáticamente en la correspondiente rama remota.

Cuando clonas un repositorio, este suele crear automáticamente una rama `master` que hace seguimiento de `origin/master`. Y es por eso que `git push` y `git pull` trabajan

directamente, sin necesidad de más argumentos. Sin embargo, puedes preparar otras ramas de seguimiento si deseas tener unas que no hagan seguimiento de ramas en origin y que no sigan a la rama master. El ejemplo más simple, es el que acabas de ver al lanzar el comando git checkout -b [rama] [nombreremoto]/[rama]. Si tienes la versión 1.6.2 de Git, o superior, puedes utilizar también el parámetro --track:

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch
refs/remotes/origin/serverfix.
Switched to a new branch "serverfix"Switched to
a new branch "serverfix"
```

Para preparar una rama local con un nombre distinto a la del remoto, puedes utilizar:

```
$ git checkout -b sf origin/serverfix
Branch sf set up to track remote branch
refs/remotes/origin/serverfix.
Switched to a new branch "sf"
```

Así, tu rama local sf va a llevar (push) y traer (pull) hacia o desde origin/serverfix.

Borrando ramas remotas

Imagina que ya has terminado con una rama remota. Es decir, tanto tu como tus colaboradores habeis completado una determinada funcionalidad y la habeis incorporado (merge) a la rama master en el remoto (o donde quiera que tengais la rama de código estable). Puedes borrar la rama remota utilizando la un tanto confusa sintaxis: git push [nombreremoto] :[rama]. Por ejemplo, si quieres borrar la rama serverfix del servidor, puedes utilizar:

```
$ git push origin :serverfix
To git@github.com:schacon/simplegit.git
 - [deleted]         serverfix
```

Y... ¡Boom!. La rama en el servidor ha desaparecido. Puedes grabarte a fuego esta página, porque necesitarás ese comando y, lo más probable es que hayas olvidado su sintaxis. Una manera de recordar este comando es dándonos

cuenta de que proviene de la sintaxis `git push [nombreremoto] [ramalocal]:[ramaremota]`. Si omites la parte `[ramalocal]`, lo que estás diciendo es: "no cojas nada de mi lado y haz con ello [ramaremota]".

6 Reorganizando el trabajo realizado

En Git tenemos dos formas de integrar cambios de una rama en otra: la fusión (merge) y la reorganización (rebase). En esta sección vas a aprender en qué consiste la reorganización, como utilizarla, por qué es una herramienta sorprendente y en qué casos no es conveniente utilizarla.

Reorganización básica

Volviendo al ejemplo anterior, en la sección sobre fusiones (ver Figura 3-27), puedes ver que has separado tu trabajo y realizado confirmaciones (commit) en dos ramas diferentes.

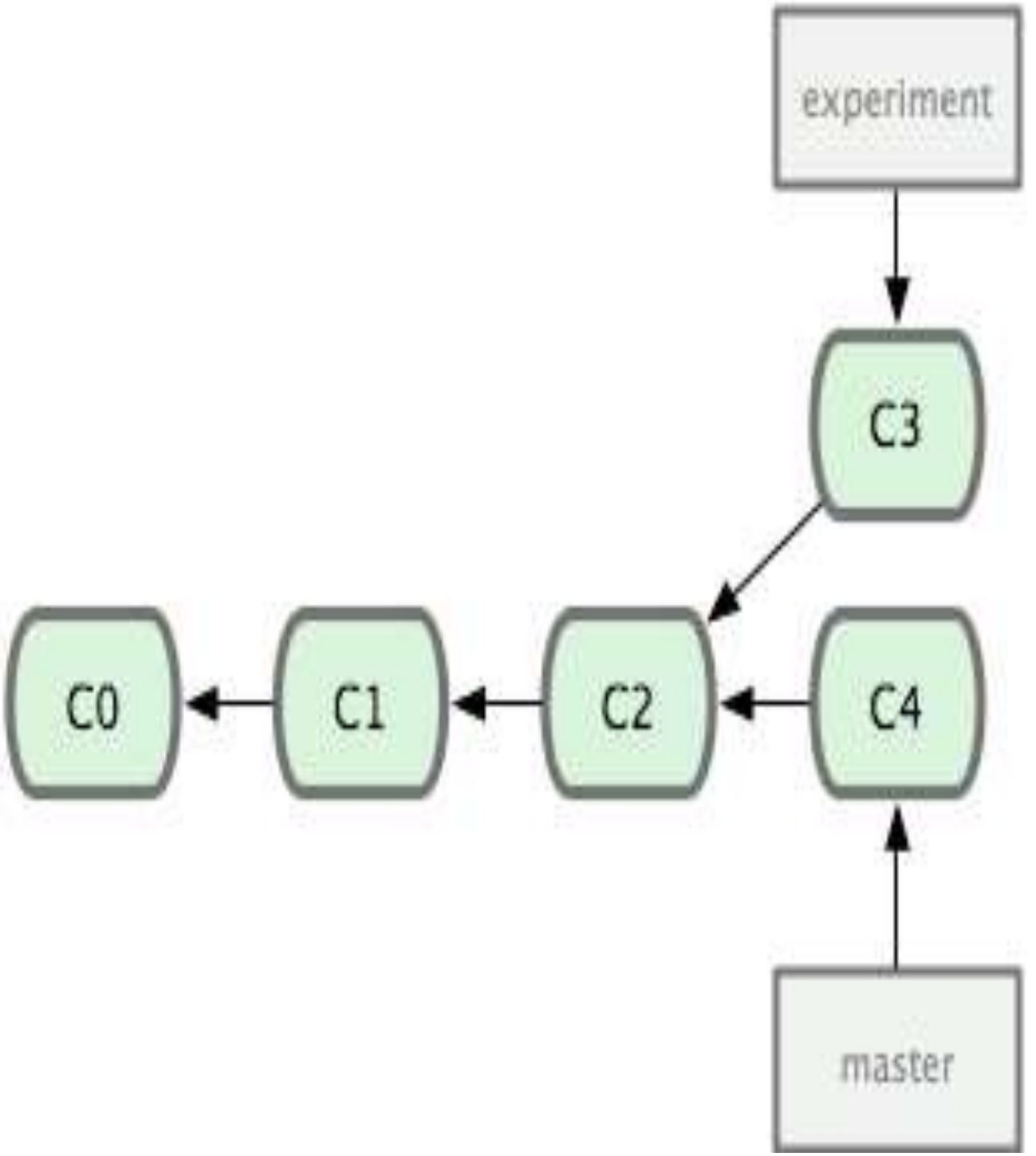


Figura 3-27. El registro de confirmaciones inicial.

La manera más sencilla de integrar ramas, tal y como hemos visto, es el comando `git merge`. Realiza una fusión a tres bandas entre las dos últimas instantáneas de cada rama (C3 y C4) y el ancestro común a ambas (C2); creando una nueva instantánea (snapshot) y la correspondiente confirmación (commit), según se muestra en la Figura 3-28.

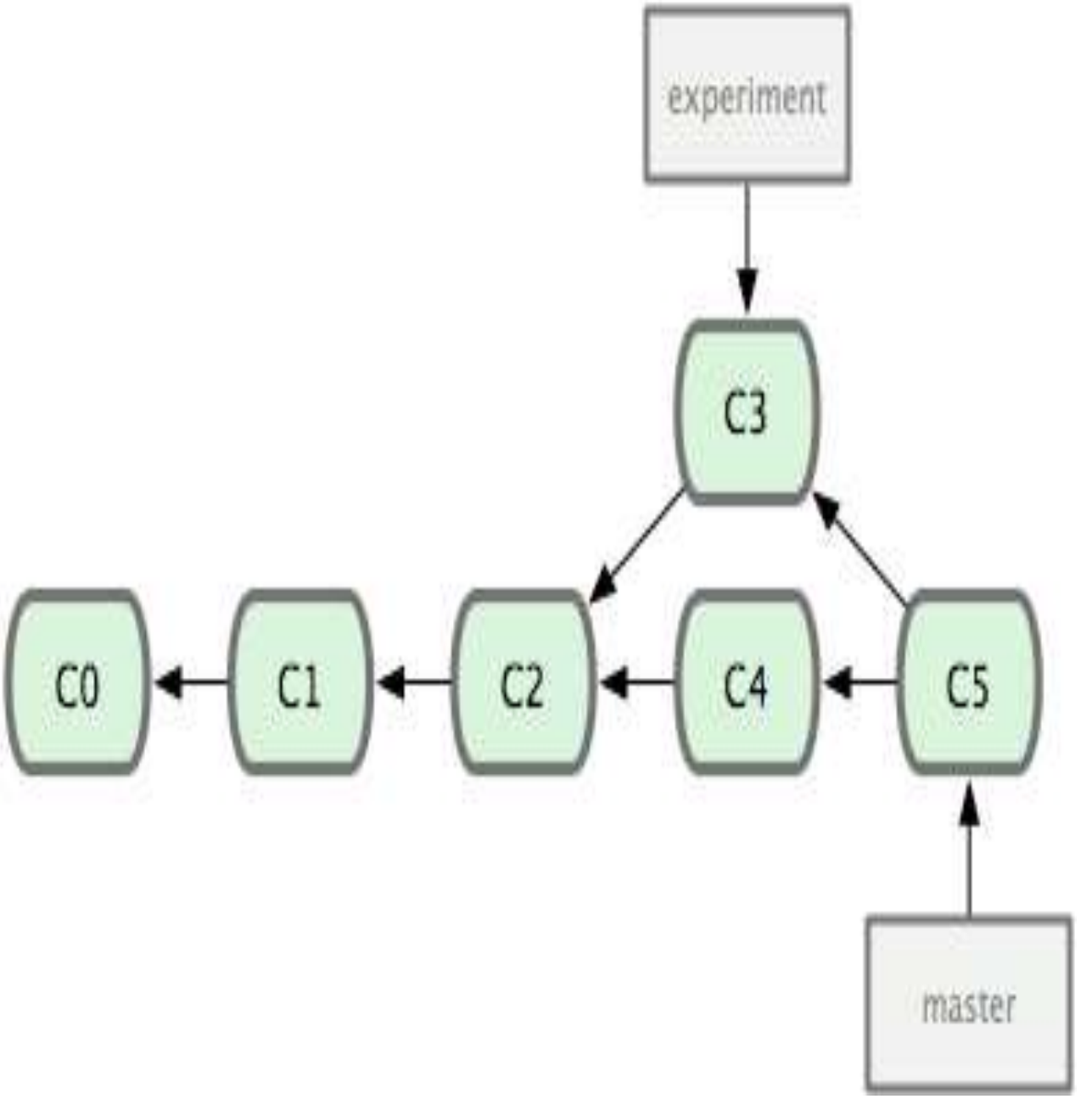


Figura 3-28. Fusionando una rama para integrar el registro de trabajos divergentes.

Aunque también hay otra forma de hacerlo: puedes coger los cambios introducidos en C3 y reaplicarlos encima de C4. Esto es lo que en Git llamamos *reorganizar*. Con el comando git rebase, puedes coger todos los cambios confirmados en una rama, y reaplicarlos sobre otra.

Por ejemplo, puedes lanzar los comandos:

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

Haciendo que Git: vaya al ancestro común de ambas ramas (donde estás actualmente y de donde quieres reorganizar), saque las diferencias introducidas por cada confirmación en la rama donde estás, guarde esas diferencias en archivos temporales,

reinicie (reset) la rama actual hasta llevarla a la misma confirmación en la rama de donde quieres reorganizar, y, finalmente, vuelva a aplicar ordenadamente los cambios. El proceso se muestra en la Figura 3-29.

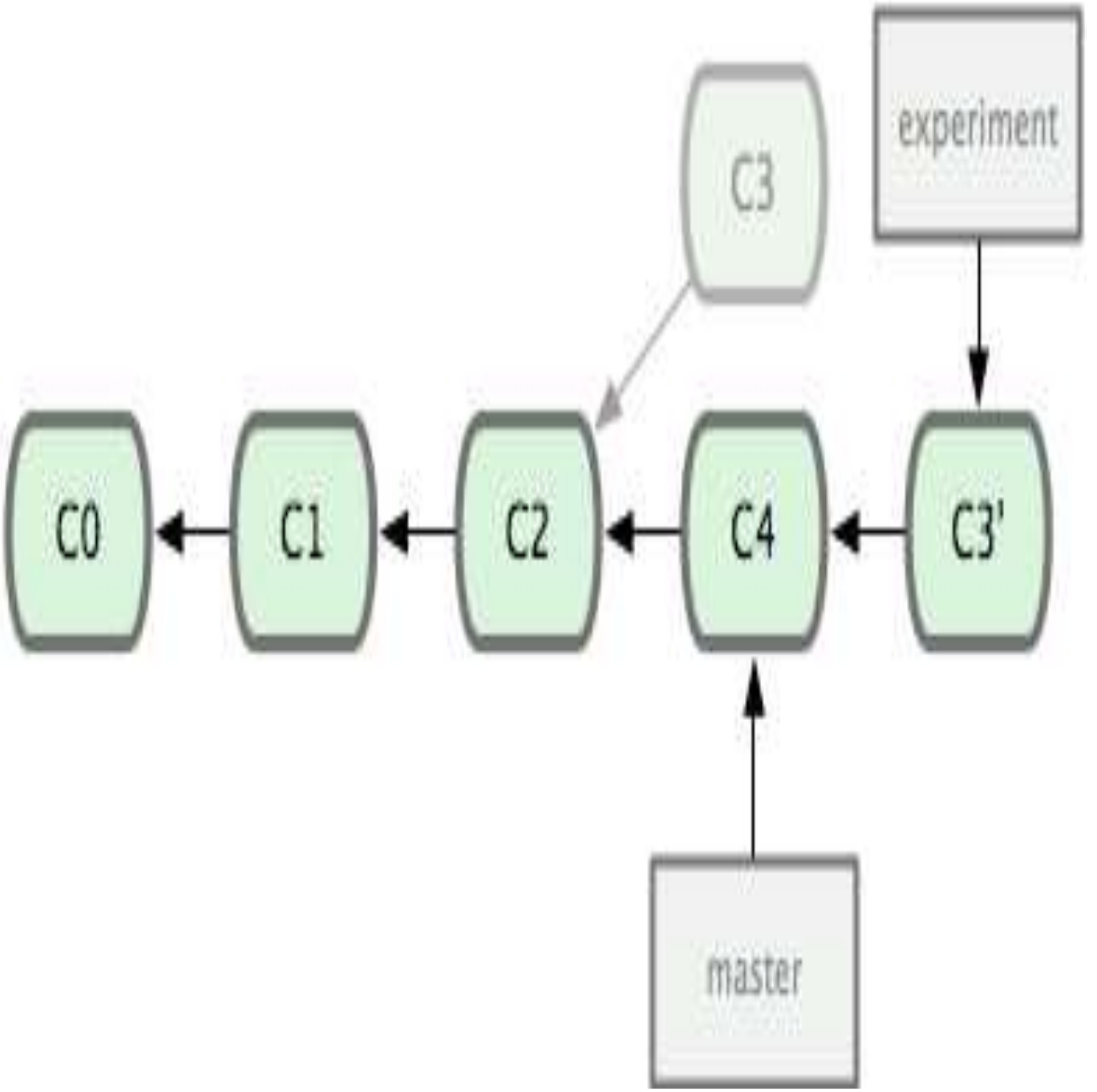


Figura 3-29. Reorganizando sobre C4 los cambios introducidos en C3.

En este momento, puedes volver a la rama master y hacer una fusión con avance rápido (fast-forward merge). (ver Figura 3-30)

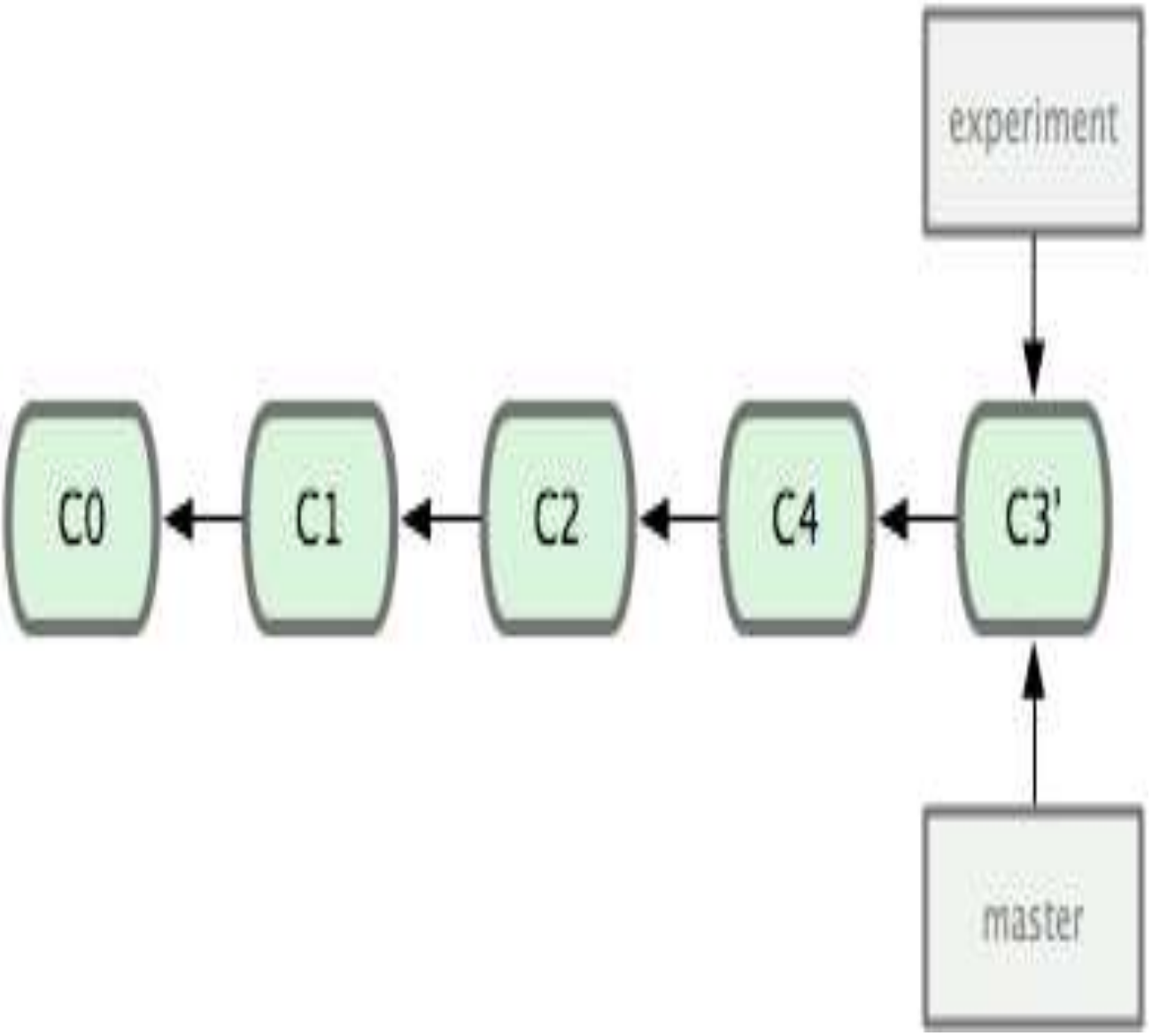


Figura 3-30. Avance rápido de la rama master.

Así, la instantánea apuntada por C3' aquí es exactamente la misma apuntada por C5 en el ejemplo de la fusión. No hay ninguna diferencia en el resultado final de la integración, pero el haberla hecho reorganizando nos deja un registro más claro. Si examinas el registro de una rama reorganizada, este aparece siempre como un registro lineal: como si todo el trabajo se hubiera realizado en series, aunque realmente se haya hecho en paralelo.

Habitualmente, optarás por esta vía cuando quieras estar seguro de que tus confirmaciones de cambio (commits) se pueden aplicar limpiamente sobre una rama remota; posiblemente, en un proyecto donde estés intentando colaborar, pero lleves tu el

mantenimiento. En casos como esos, puedes trabajar sobre una rama y luego reorganizar lo realizado en la rama origin/master cuando lo tengas todo listo para enviarlo al proyecto principal. De esta forma, la persona que mantiene el proyecto no necesitará hacer ninguna integración con tu trabajo; le bastará con un avance rápido o una incorporación limpia.

Cabe destacar que la instantánea (snapshot) apuntada por la confirmación (commit) final, tanto si es producto de una reorganización (rebase) como si lo es de una fusión (merge), es exactamente la misma instantánea. Lo único diferente es el registro. La reorganización vuelve a aplicar cambios de una rama de trabajo sobre otra rama, en el mismo orden en que fueron introducidos en la primera. Mientras que la fusión combina entre sí los dos puntos finales de ambas ramas.

Algunas otras reorganizaciones interesantes

También puedes aplicar una reorganización (rebase) sobre otra cosa además de sobre la rama de reorganización. Por ejemplo, sea un registro como el de la Figura 3-31. Has ramificado a una rama puntual (server) para añadir algunas funcionalidades al proyecto, y luego has confirmado los cambios. Después, vuelves a la rama original para hacer algunos cambios en la parte cliente (rama client), y confirmas también esos cambios. Por último, vuelves sobre la rama server y haces algunos cambios más.

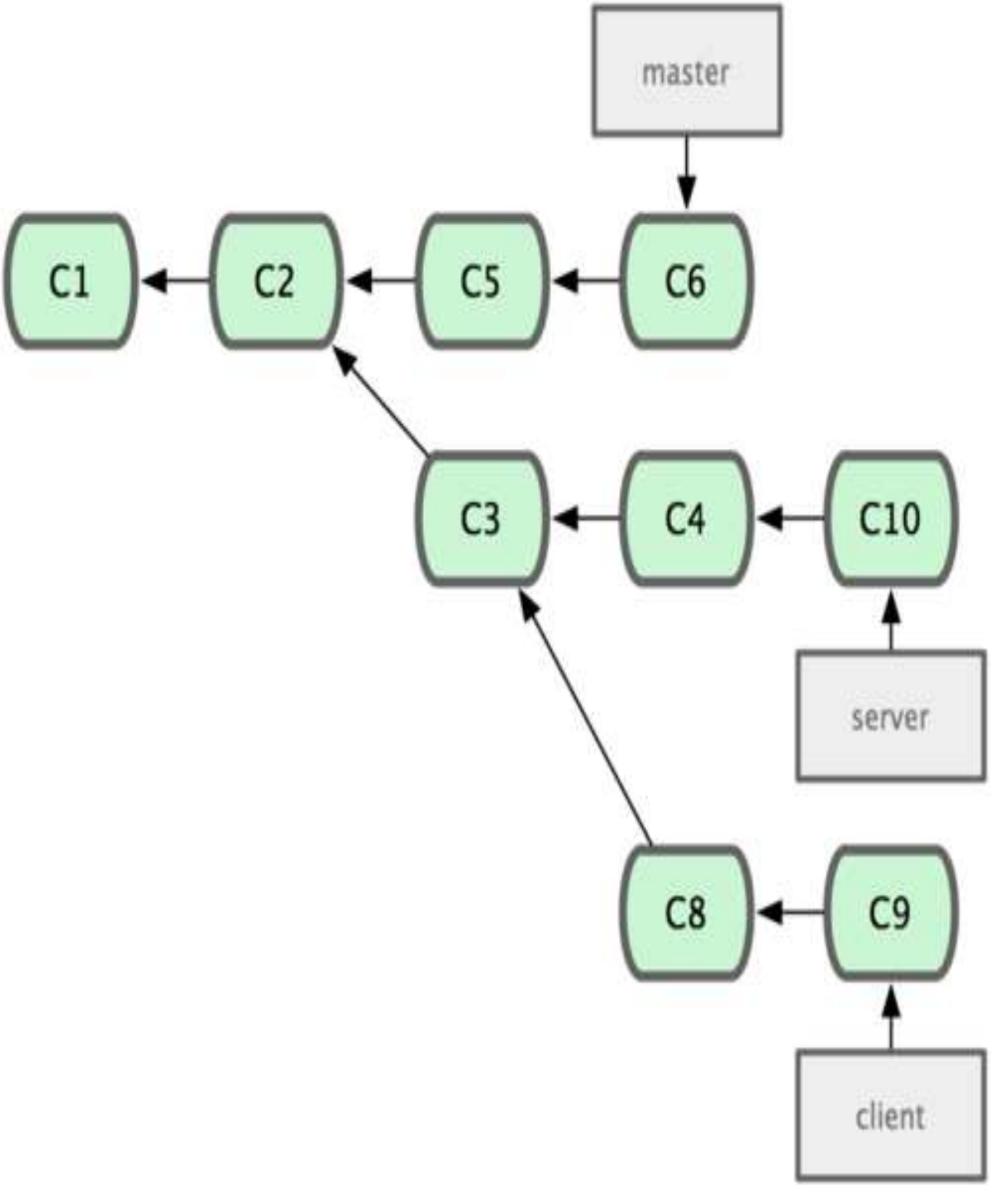


Figura 3-31. Un registro con una rama puntual sobre otra rama puntual.

Imagina que decides incorporar tus cambios de la parte cliente sobre el proyecto principal, para hacer un lanzamiento de versión; pero no quieres lanzar aún los cambios de la parte server porque no están aún suficientemente probados. Puedes coger los cambios del cliente que no están en server (C8 y C9), y reaplicarlos sobre tu rama principal usando la opción `--onto` del comando `git rebase`:

```
$ git rebase --onto master server client
```

Esto viene a decir: "Activa la rama `client`, averigua los cambios desde el ancestro común entre las ramas `client` y `server`, y aplicarlos en la rama `master`. Puede parecer un poco complicado, pero los resultados, mostrados en la Figura 3-32, son realmente interesantes.

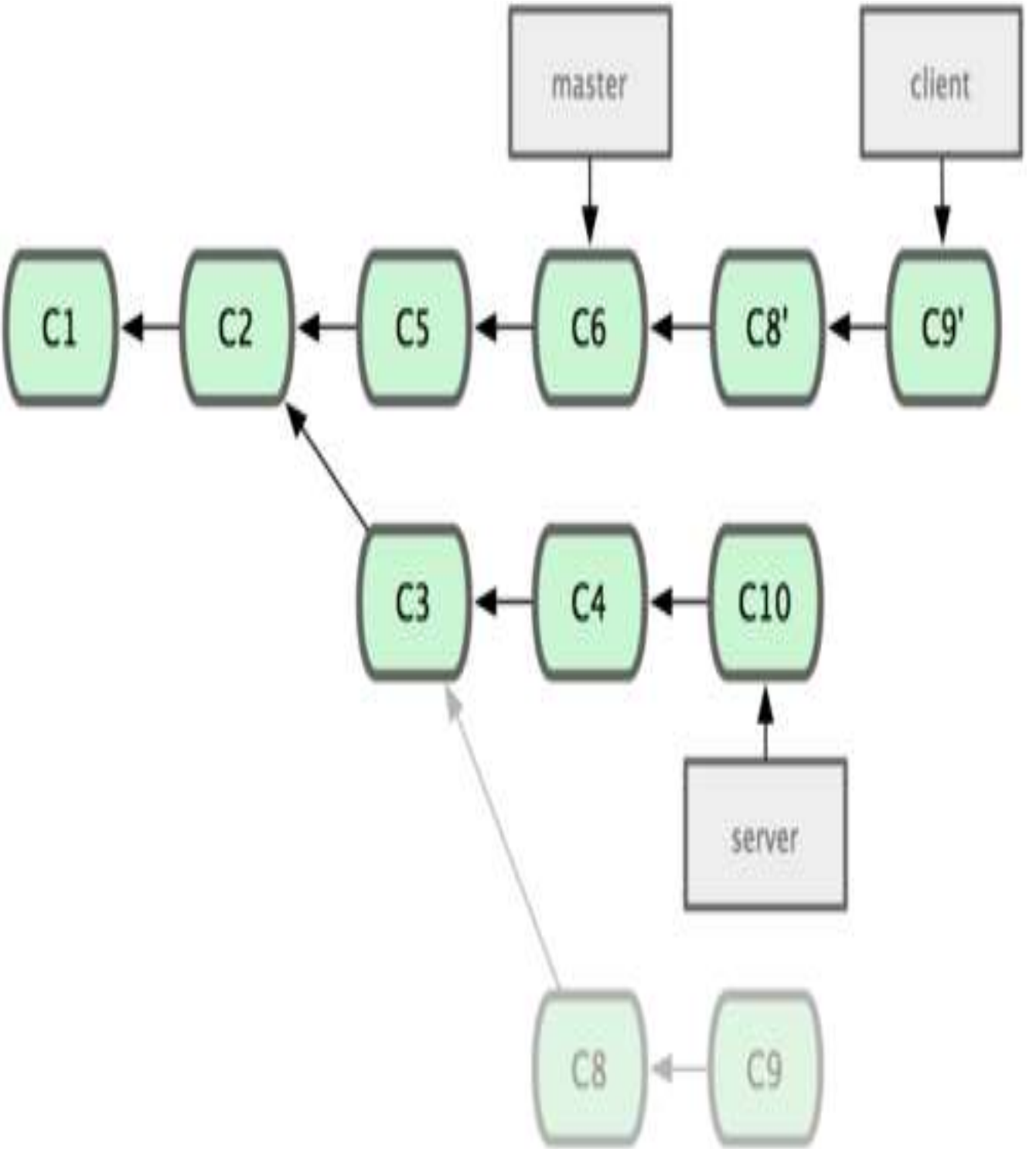


Figura 3-32. Reorganizando una rama puntual fuera de otra rama puntual.

Y, tras esto, ya puedes avanzar la rama principal (ver Figura 3-33):

```
$ git checkout master
$ git merge client
```

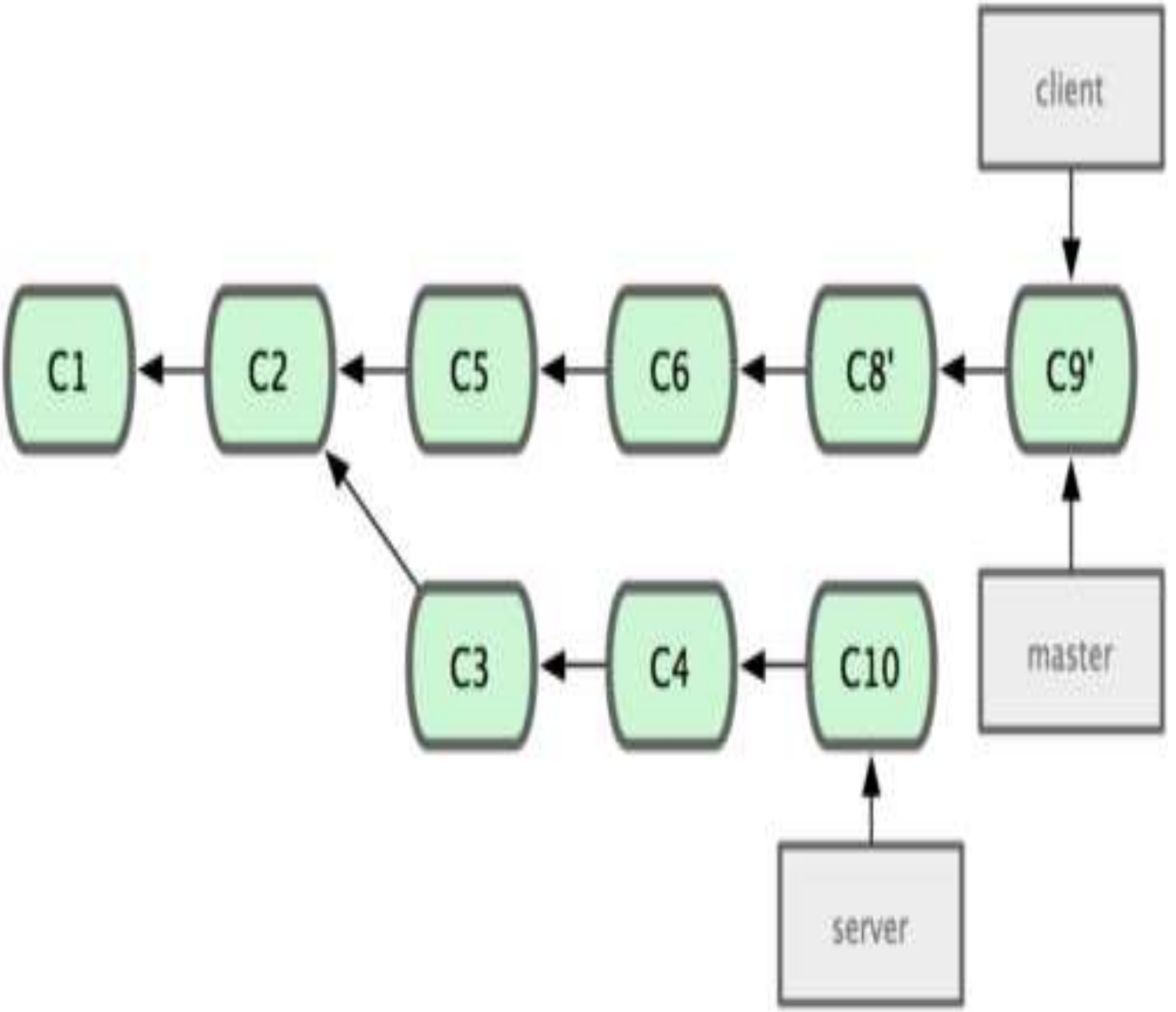


Figura 3-33. Avance rápido de tu rama `master`, para incluir los cambios de la rama `client`.

Ahora supongamos que decides traerlos (pull) también sobre tu rama server. Puedes reorganizar (rebase) la rama server sobre la rama master sin necesidad siquiera de comprobarlo previamente, usando el comando git rebase [ramabase] [ramapuntual]. El cual activa la rama puntual (server en este caso) y la aplica sobre la rama base (master en este caso):

```
$ git rebase master server
```

Esto vuelca el trabajo de server sobre el de master, tal y como se muestra en la Figura 3-34.

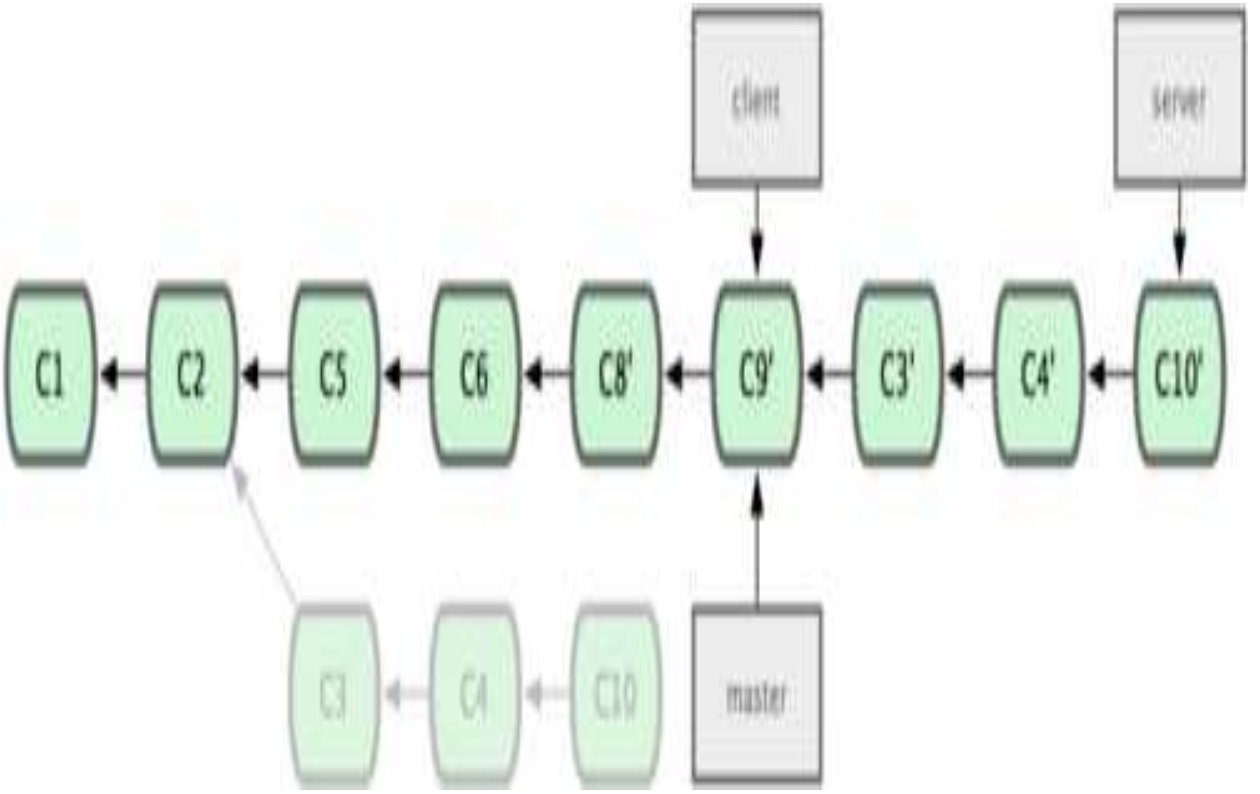


Figura 3-34. Reorganizando la rama server sobre la rama branch.

Después, puedes avanzar rápidamente la rama base (master):

```
$ git checkout master
$ git merge server
```

Y por último puedes eliminar las ramas client y server porque ya todo su contenido ha sido integrado y no las vas

a necesitar más. Dejando tu registro tras todo este proceso tal y como se muestra en la Figura 3-35:

```
$ git branch -d client
$ git branch -d server
```

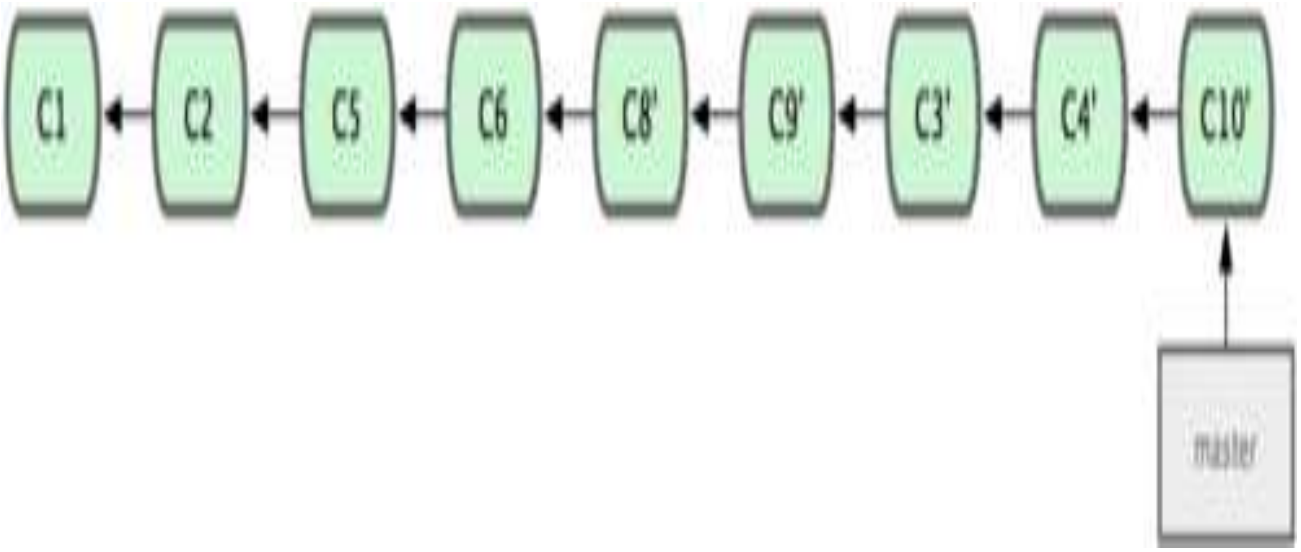


Figura 3-35. Registro final de confirmaciones de cambio.

Los peligros de la reorganización

Ahh..., pero la dicha de la reorganización no la alcanzamos sin sus contrapartidas:

Nunca reorganices confirmaciones de cambio (commits) que hayas enviado (push) a un repositorio público.

Siguiendo esta recomendación, no tendrás problemas. Pero si no la sigues, la gente te odiará y serás despreciado por tus familiares y amigos.

Cuando reorganizas algo, estás abandonando las confirmaciones de cambio ya creadas y estás creando unas nuevas; que son similares, pero diferentes. Si envias (push) confirmaciones (commits) a alguna parte, y otros las recogen (pull) de allí. Y después vas tu y las reescribes con `git rebase` y las vuelves a enviar (push) de nuevo. Tus colaboradores tendrán que

refusionar (re-merge) su trabajo y todo se volverá tremendamente complicado cuando intentes recoger (pull) su trabajo de vuelta sobre el tuyo.

Vamos a verlo con un ejemplo. Imagínate que haces un clon desde un servidor central, y luego trabajas sobre él. Tu registro de cambios puede ser algo como lo de la Figura 3-36.

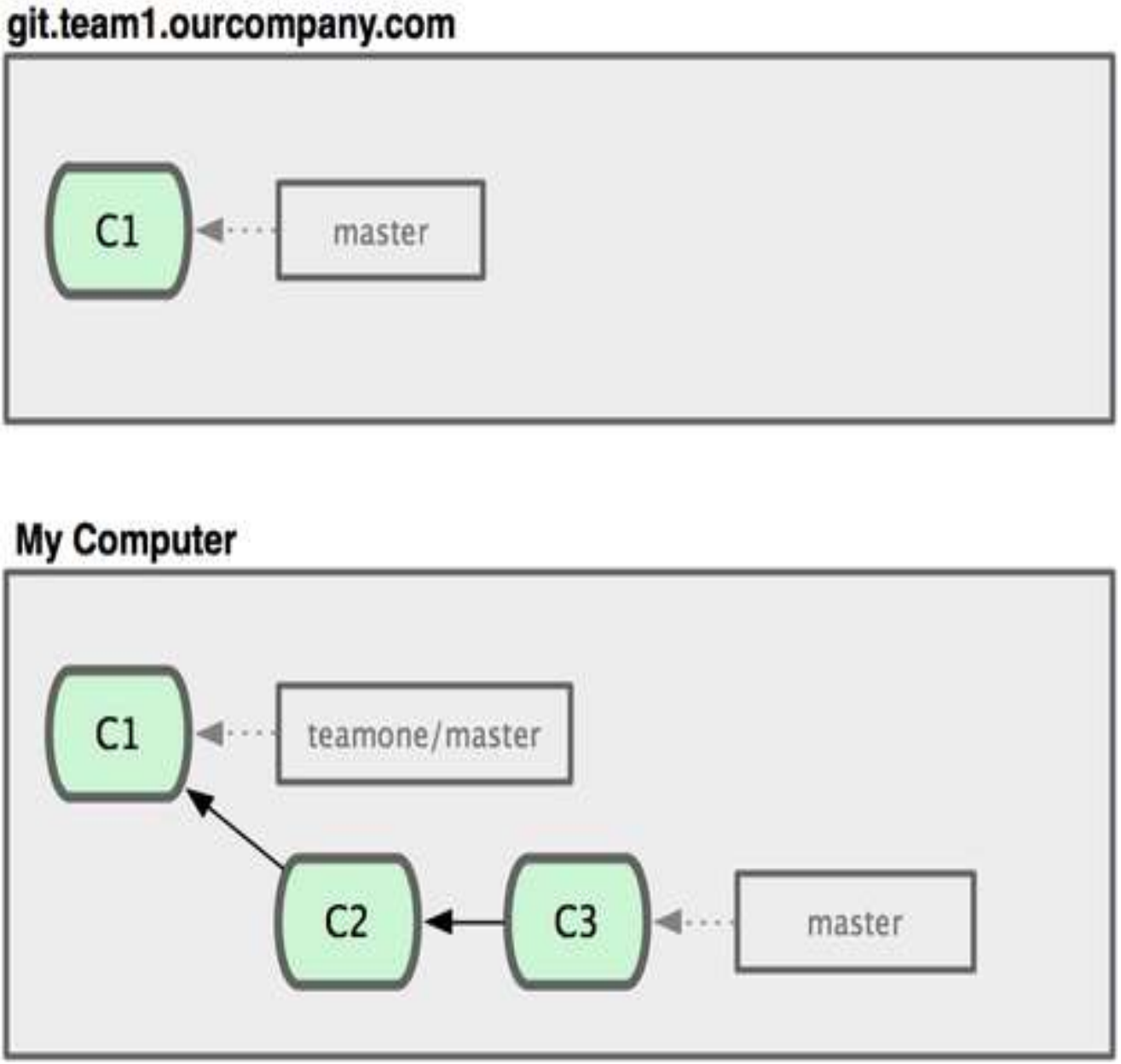


Figura 3-36. Clonar un repositorio y trabajar sobre él.

Ahora, otra persona trabaja también sobre ello, realiza una fusión (merge) y lleva (push) su trabajo al servidor

central. Tu te traes (fetch) sus trabajos y los fusionas (merge) sobre una nueva rama en tu trabajo. Quedando tu registro de confirmaciones como en la Figura 3-37.

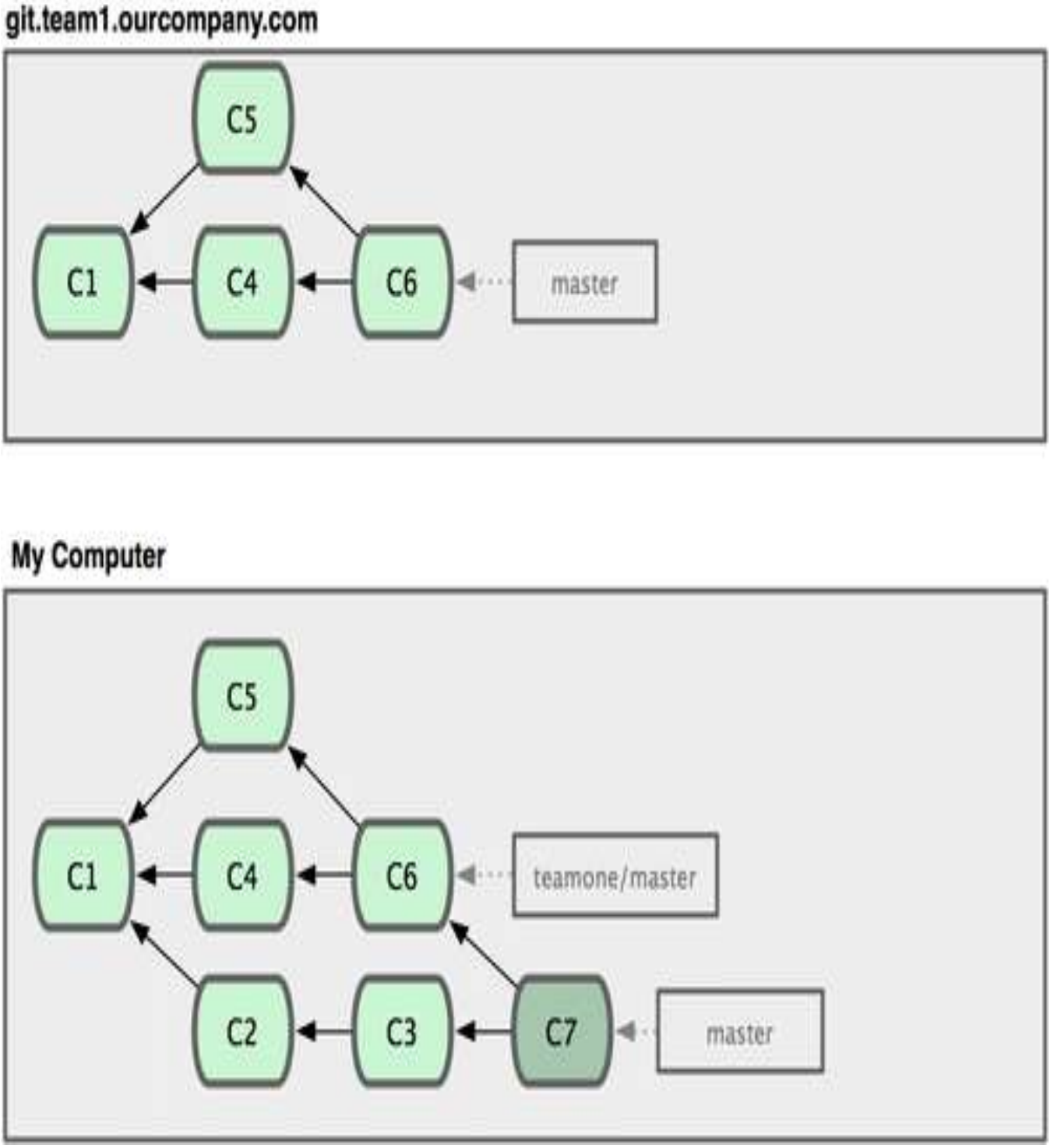


Figura 3-37. Traer (fetch) algunas confirmaciones de cambio (commits) y fusionarlas (merge) sobre tu trabajo.

A continuación, la persona que había llevado cambios al servidor central decide retroceder y reorganizar su trabajo; haciendo un `git push --force` para sobrescribir el registro en el servidor. Tu te traes (fetch) esos nuevos cambios desde el servidor.

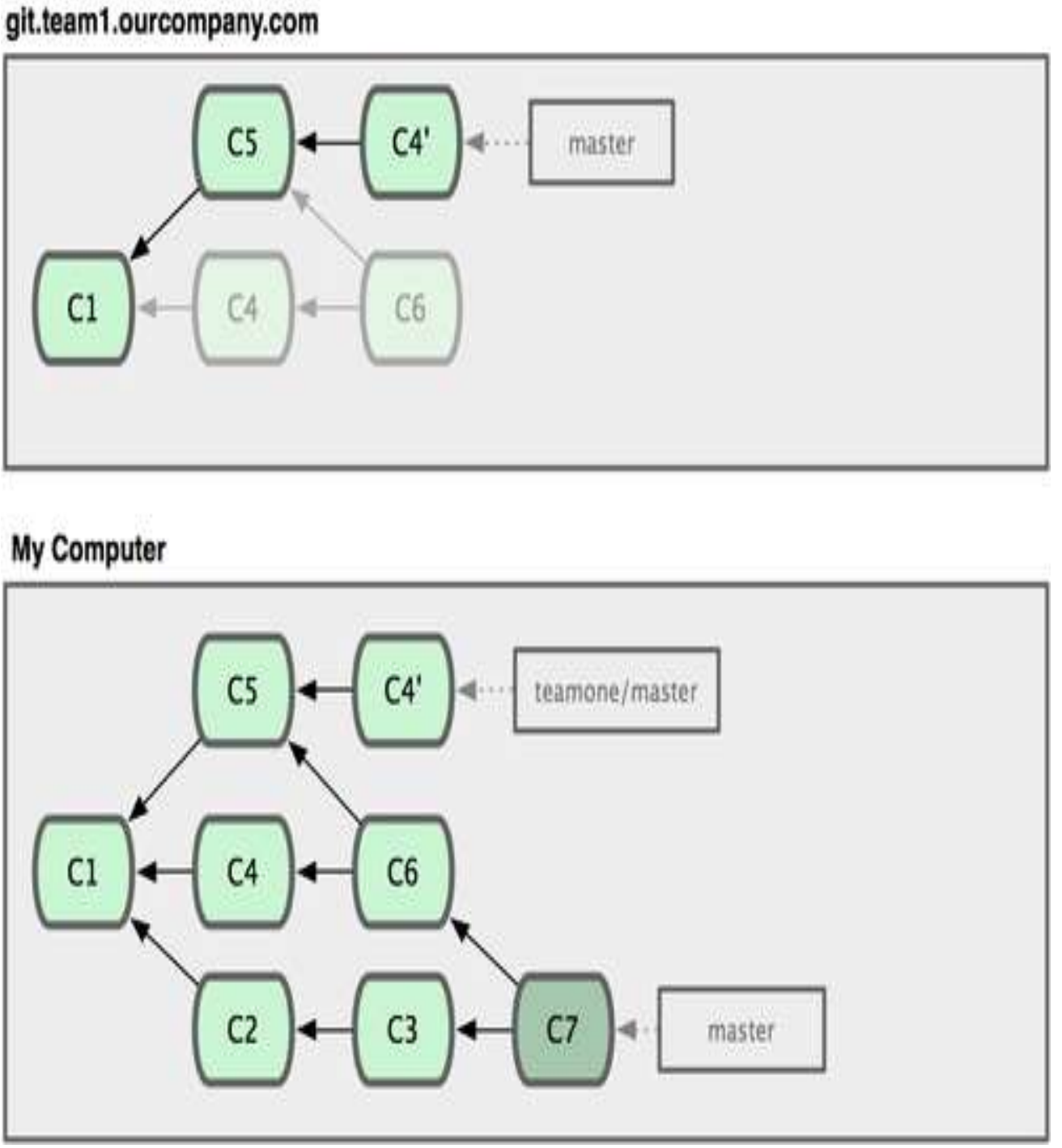


Figura 3-38. Alguien envií (push) confirmaciones (commits) reorganizadas, abandonando las

confirmaciones en las que tu habías basado tu trabajo.

En ese momento, tu te ves obligado a fusionar (merge) tu trabajo de nuevo, aunque creias que ya lo habias hecho antes. La reorganización cambia los resumenes (hash) SHA-1 de esas confirmaciones (commits), haciendo que Git se crea que son nuevas confirmaciones. Cuando realmente tu ya tenias el trabajo de C4 en tu registro.

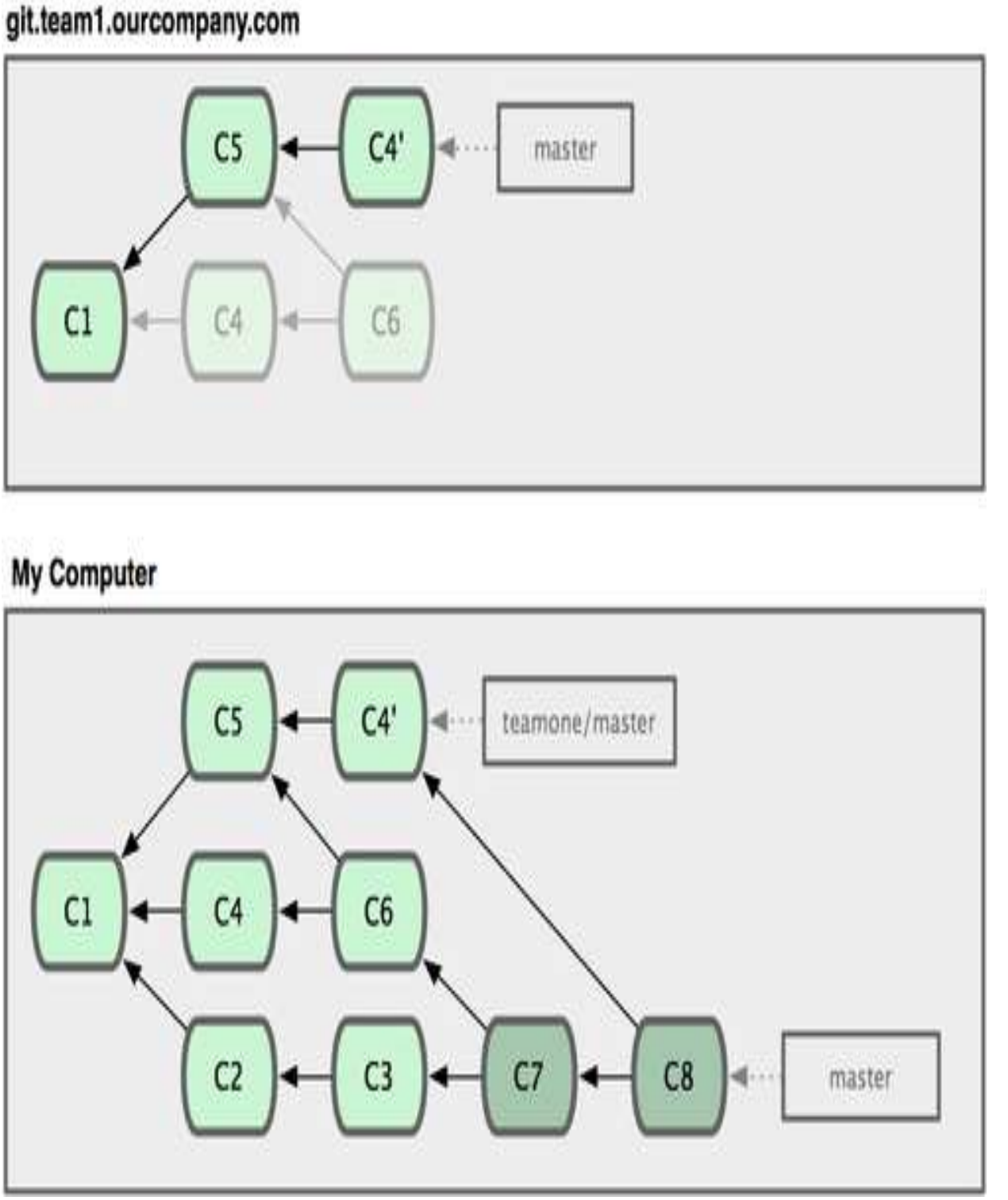


Figura 3-39. Vuelves a fusionar el mismo trabajo en una nueva fusión confirmada.

Te ves obligado a fusionar (merge) ese trabajo en algún punto, para poder seguir adelante con otros desarrollos en el futuro. Tras todo esto, tu registro de confirmaciones de cambio (commit history) contendrá tanto la confirmación C4 como la C4'; teniendo ambas el mismo contenido y el mismo mensaje de confirmación. Si lanzas un `git log` en un registro como este, verás dos confirmaciones con el mismo autor, misma fecha y mismo mensaje. Lo que puede llevar a confusiones. Es más, si luego tu envías (push) ese registro de vuelta al servidor, vas a introducir todas esas confirmaciones reorganizadas en el servidor central. Lo que puede confundir aún más a la gente.

Si solo usas la reorganización como una vía para hacer limpieza y organizar confirmaciones de cambio antes de enviarlas, y si únicamente reorganizas confirmaciones que nunca han sido

públicas. Entonces no tendrás problemas. Si, por el contrario, reorganizas confirmaciones que alguna vez han sido públicas y otra gente ha basado su trabajo en ellas. Entonces estarás en un aprieto.

.6 Reorganizando el trabajo realizado

En Git tenemos dos formas de integrar cambios de una rama en otra: la fusión (merge) y la reorganización (rebase). En esta sección vas a aprender en qué consiste la reorganización, como utilizarla, por qué es una herramienta sorprendente y en qué casos no es conveniente utilizarla.

Reorganización básica

Volviendo al ejemplo anterior, en la sección sobre fusiones (ver Figura 3-27), puedes ver que has separado tu trabajo y realizado confirmaciones (commit) en dos ramas diferentes.

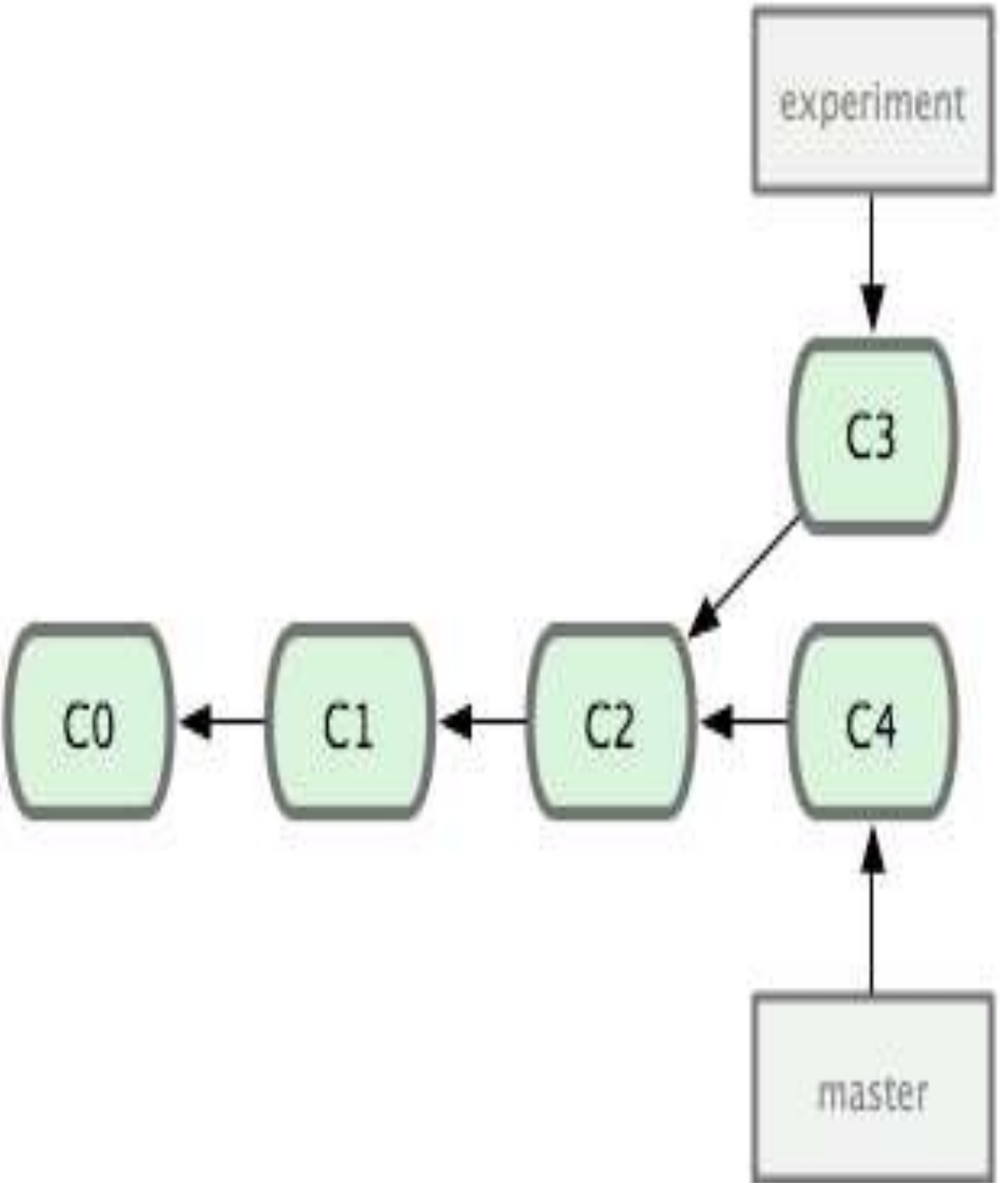


Figura 3-27. El registro de confirmaciones inicial.

La manera más sencilla de integrar ramas, tal y como hemos visto, es el comando `git merge`. Realiza una fusión a tres bandas entre las dos últimas instantáneas de cada rama (C3 y C4) y el ancestro común a ambas (C2); creando una nueva instantánea (snapshot) y la correspondiente confirmación (commit), según se muestra en la Figura 3-28.

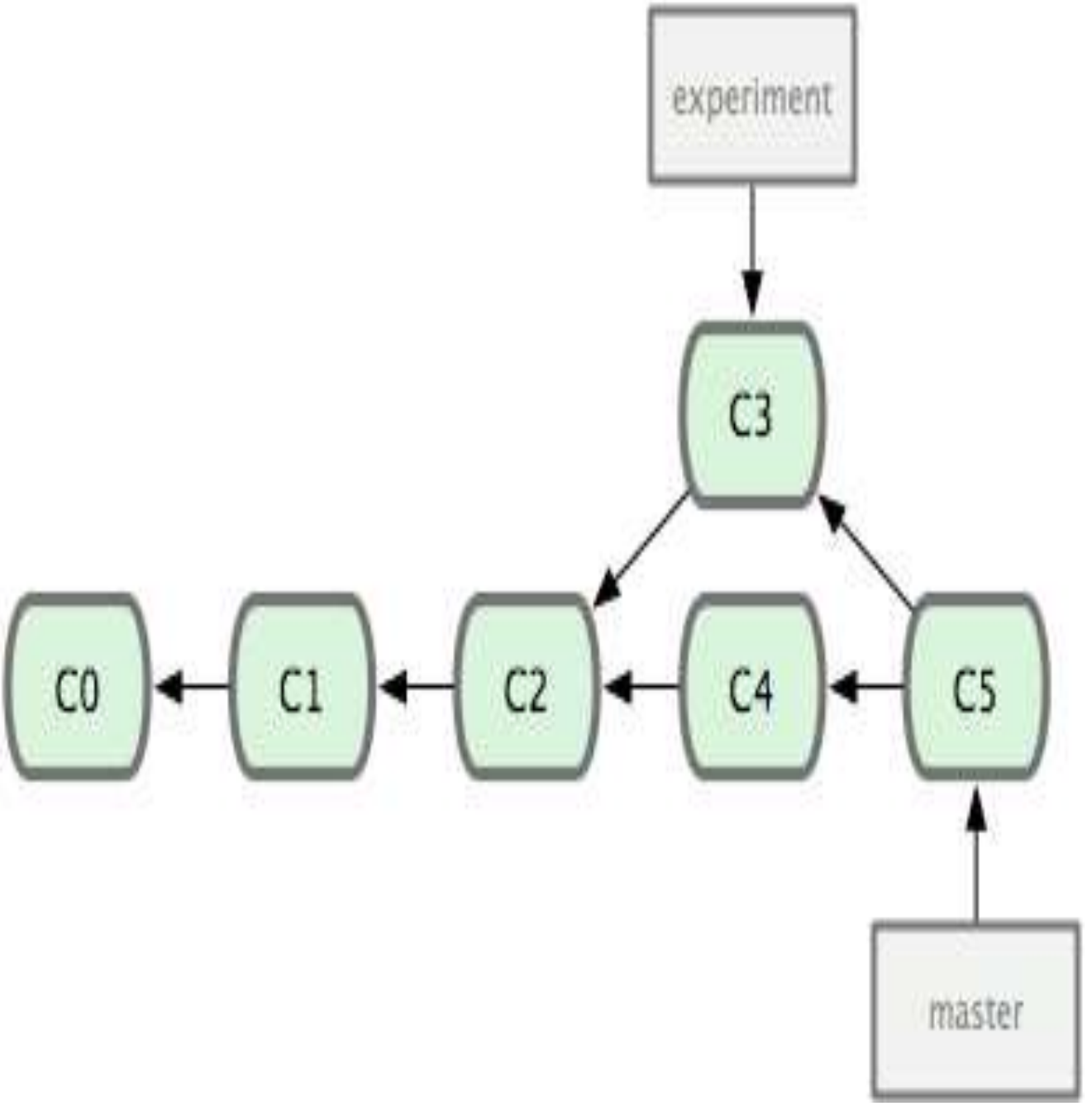


Figura 3-28. Fusionando una rama para integrar el registro de trabajos divergentes.

Aunque también hay otra forma de hacerlo: puedes coger los cambios introducidos en C3 y reaplicarlos encima de C4. Esto es lo que en Git llamamos *reorganizar*. Con el comando git rebase, puedes coger todos los cambios confirmados en una rama, y reaplicarlos sobre otra.

Por ejemplo, puedes lanzar los comandos:

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

Haciendo que Git: vaya al ancestro común de ambas ramas (donde estás actualmente y de donde quieres reorganizar), saque las diferencias introducidas por cada confirmación en la rama donde estás, guarde esas diferencias en archivos temporales,

reinicie (reset) la rama actual hasta llevarla a la misma confirmación en la rama de donde quieres reorganizar, y, finalmente, vuelva a aplicar ordenadamente los cambios. El proceso se muestra en la Figura 3-29.

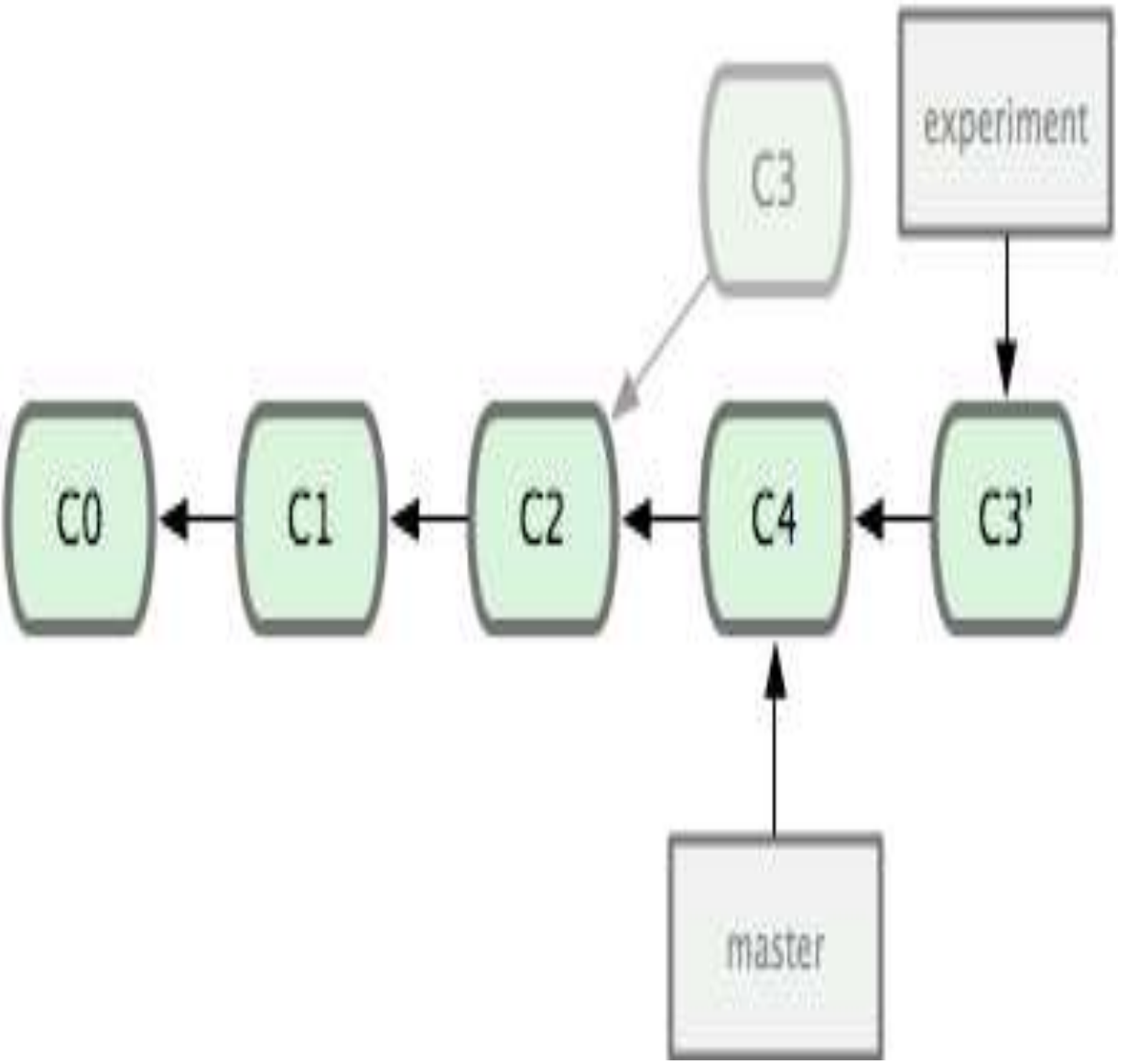


Figura 3-29. Reorganizando sobre C4 los cambios introducidos en C3.

En este momento, puedes volver a la rama master y hacer una fusión con avance rápido (fast-forward merge). (ver Figura 3-30)

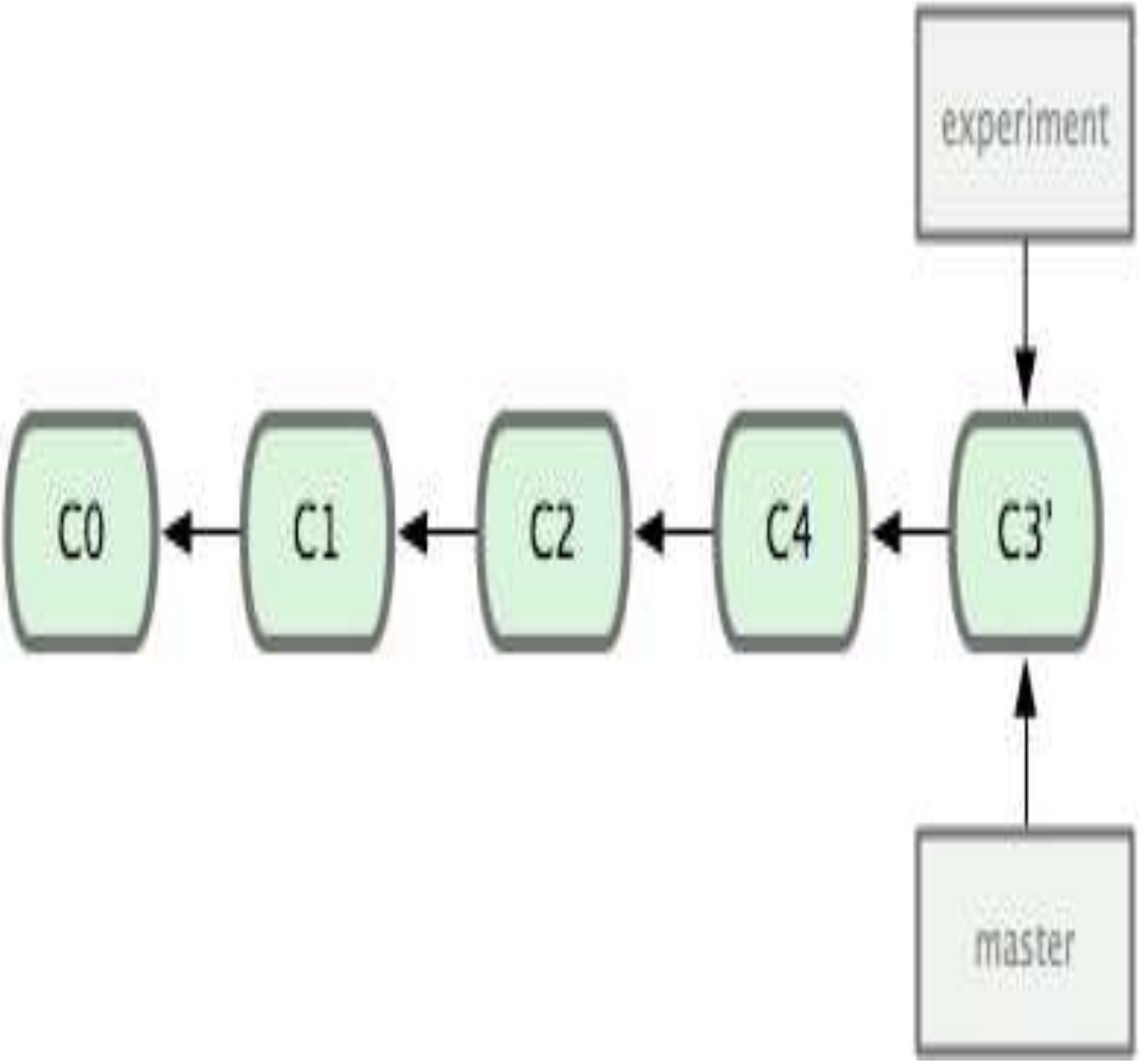


Figura 3-30. Avance rápido de la rama master.

Así, la instantánea apuntada por C3' aquí es exactamente la misma apuntada por C5 en el ejemplo de la fusión. No hay ninguna diferencia en el resultado final

de la integración, pero el haberla hecho reorganizando nos deja un registro más claro. Si examinas el registro de una rama reorganizada, este aparece siempre como un registro lineal: como si todo el trabajo se hubiera realizado en series, aunque realmente se haya hecho en paralelo.

Habitualmente, optarás por esta vía cuando quieras estar seguro de que tus confirmaciones de cambio (commits) se pueden aplicar limpiamente sobre una rama remota; posiblemente, en un proyecto donde estés intentando colaborar, pero lleves tu el mantenimiento. En casos como esos, puedes trabajar sobre una rama y luego reorganizar lo realizado en la rama `origin/master` cuando lo tengas todo listo para enviarlo al proyecto principal. De esta forma, la persona que mantiene el proyecto no necesitará hacer ninguna integración con tu trabajo; le bastará

con un avance rápido o una incorporación limpia.

Cabe destacar que la instantánea (snapshot) apuntada por la confirmación (commit) final, tanto si es producto de una reorganización (rebase) como si lo es de una fusión (merge), es exactamente la misma instantánea. Lo único diferente es el registro. La reorganización vuelve a aplicar cambios de una rama de trabajo sobre otra rama, en el mismo orden en que fueron introducidos en la primera. Mientras que la fusión combina entre sí los dos puntos finales de ambas ramas.

Algunas otras reorganizaciones interesantes

También puedes aplicar una reorganización (rebase) sobre otra cosa además de sobre la rama de reorganización. Por ejemplo, sea un registro como el de la Figura 3-31. Has ramificado a una rama puntual (server) para añadir algunas funcionalidades al proyecto, y luego has confirmado los cambios. Después, vuelves a la rama original para hacer algunos cambios en la parte cliente (rama client), y confirmas también esos cambios. Por último, vuelves sobre la rama server y haces algunos cambios más.

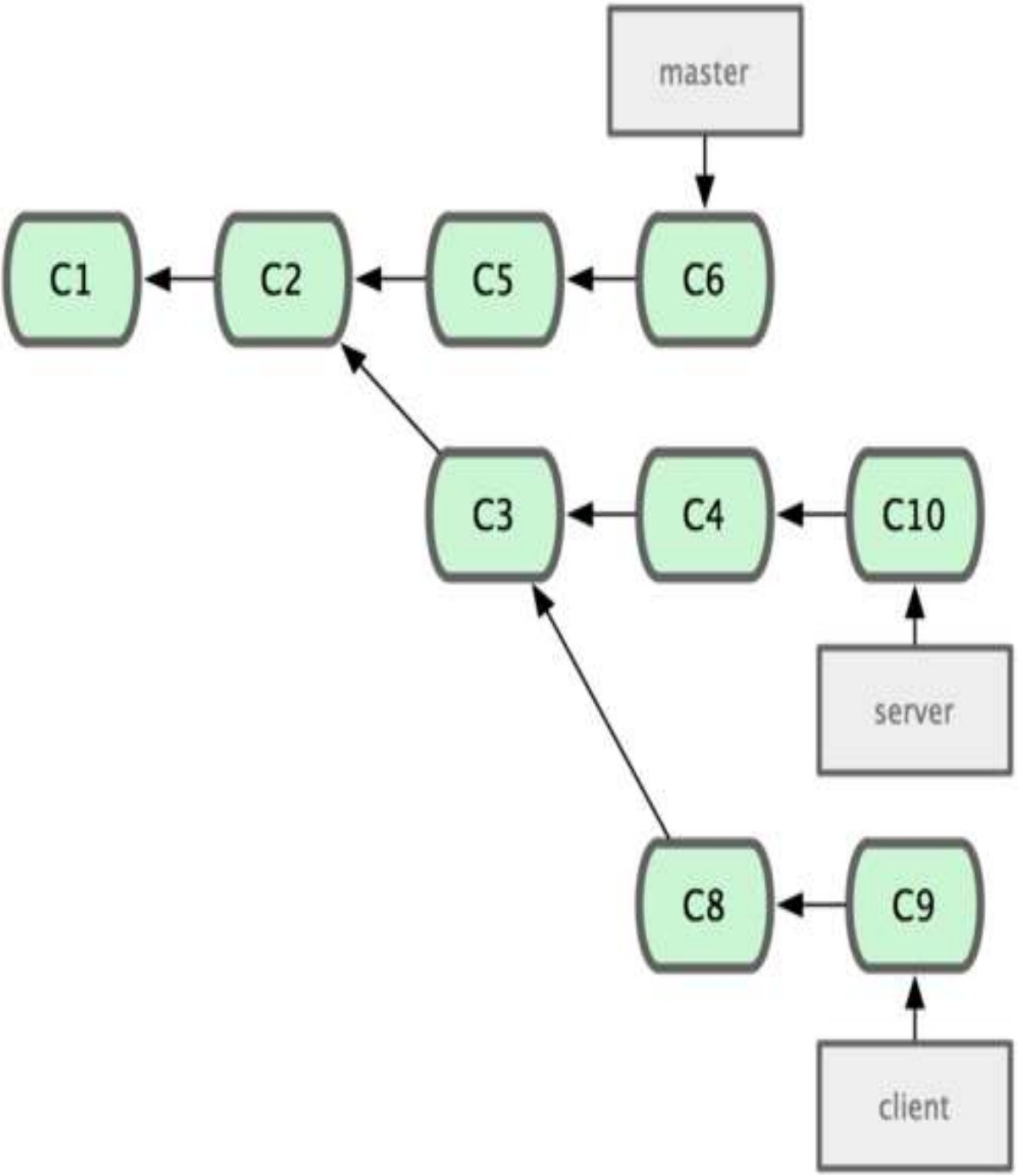


Figura 3-31. Un registro con una rama puntual sobre otra rama puntual.

Imagina que decides incorporar tus cambios de la parte cliente sobre el proyecto principal, para hacer un lanzamiento de versión; pero no quieres lanzar aún los cambios de la parte server porque no están aún suficientemente probados. Puedes coger los cambios del cliente que no están en server (C8 y C9), y reaplicarlos sobre tu rama principal usando la opción `--onto` del comando `git rebase`:

```
$ git rebase --onto master server client
```

Esto viene a decir: "Activa la rama `client`, averigua los cambios desde el ancestro común entre las ramas `client` y `server`, y aplicarlos en la rama `master`. Puede parecer un poco complicado, pero los resultados, mostrados en la Figura 3-32, son realmente interesantes.

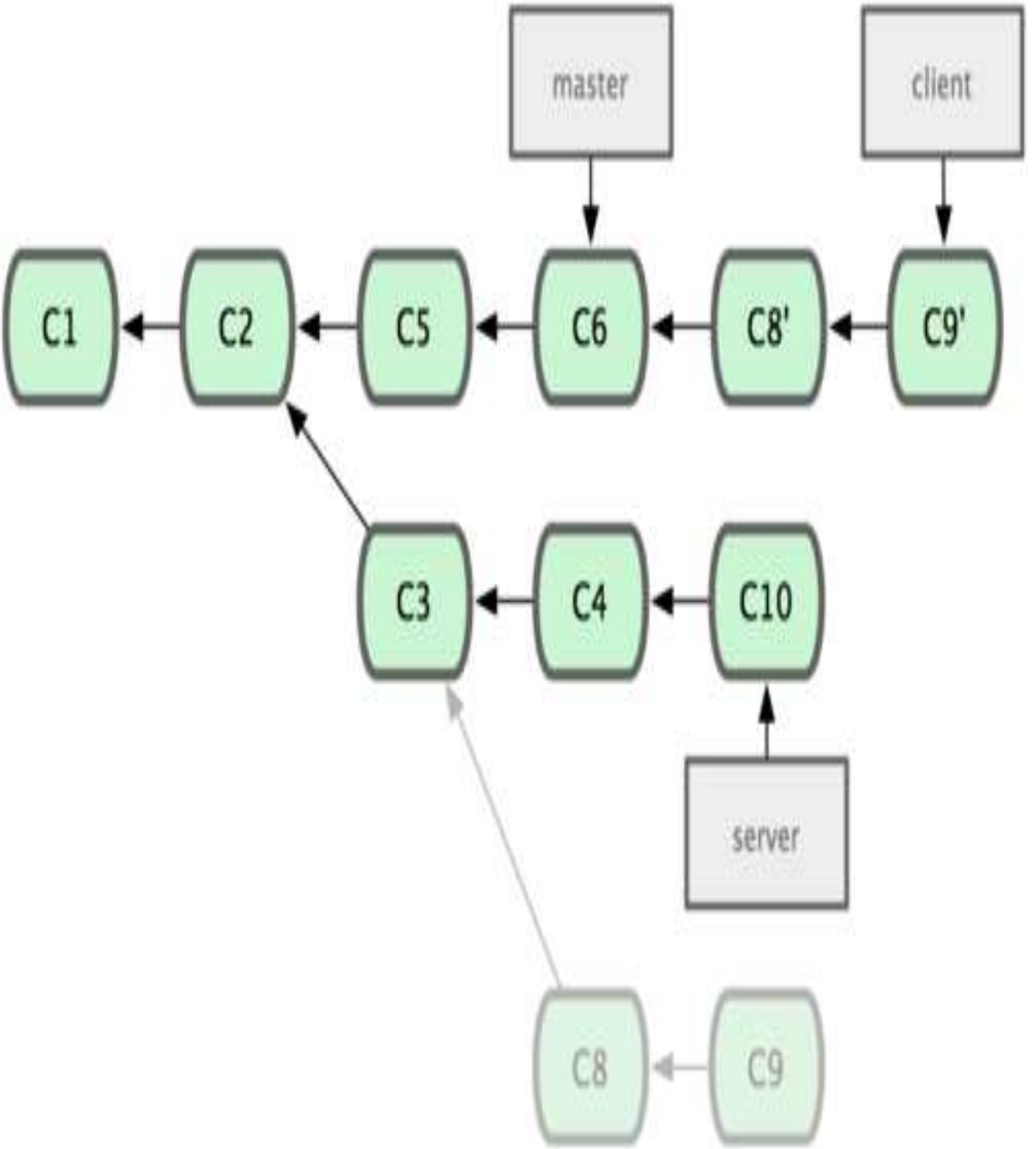


Figura 3-32. Reorganizando una rama puntual fuera de otra rama puntual.

Y, tras esto, ya puedes avanzar la rama principal (ver Figura 3-33):

```
$ git checkout master
$ git merge client
```

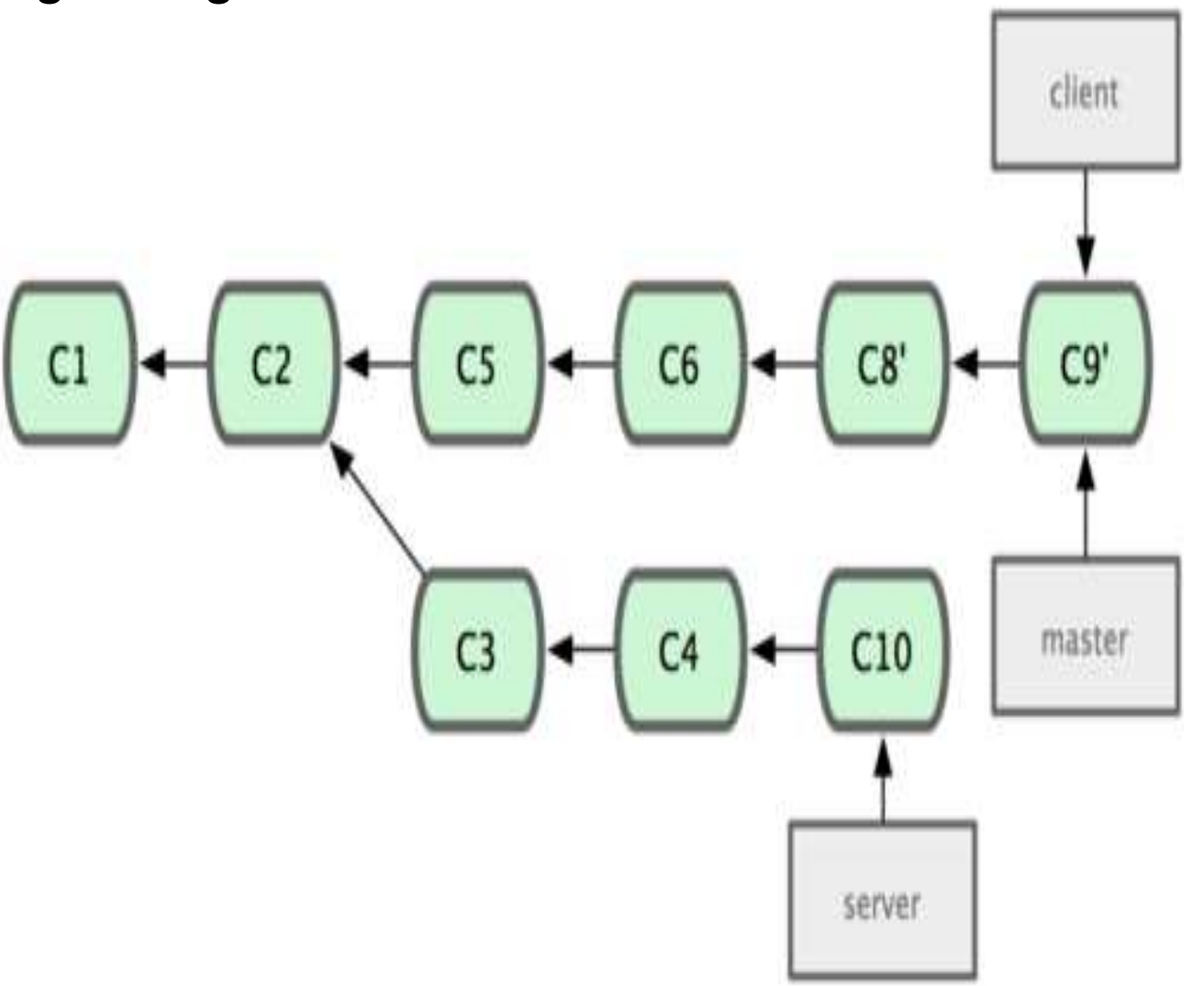


Figura 3-33. Avance rápido de tu rama master, para incluir los cambios de la rama client.

Ahora supongamos que decides traerlos (pull) también sobre tu rama server. Puedes reorganizar (rebase) la rama server sobre la rama master sin necesidad siquiera de comprobarlo previamente, usando el comando git rebase [ramabase] [ramapuntual]. El cual activa la rama puntual (server en este caso) y la aplica sobre la rama base (master en este caso):

```
$ git rebase master server
```

Esto vuelca el trabajo de server sobre el de master, tal y como se muestra en la Figura 3-34.

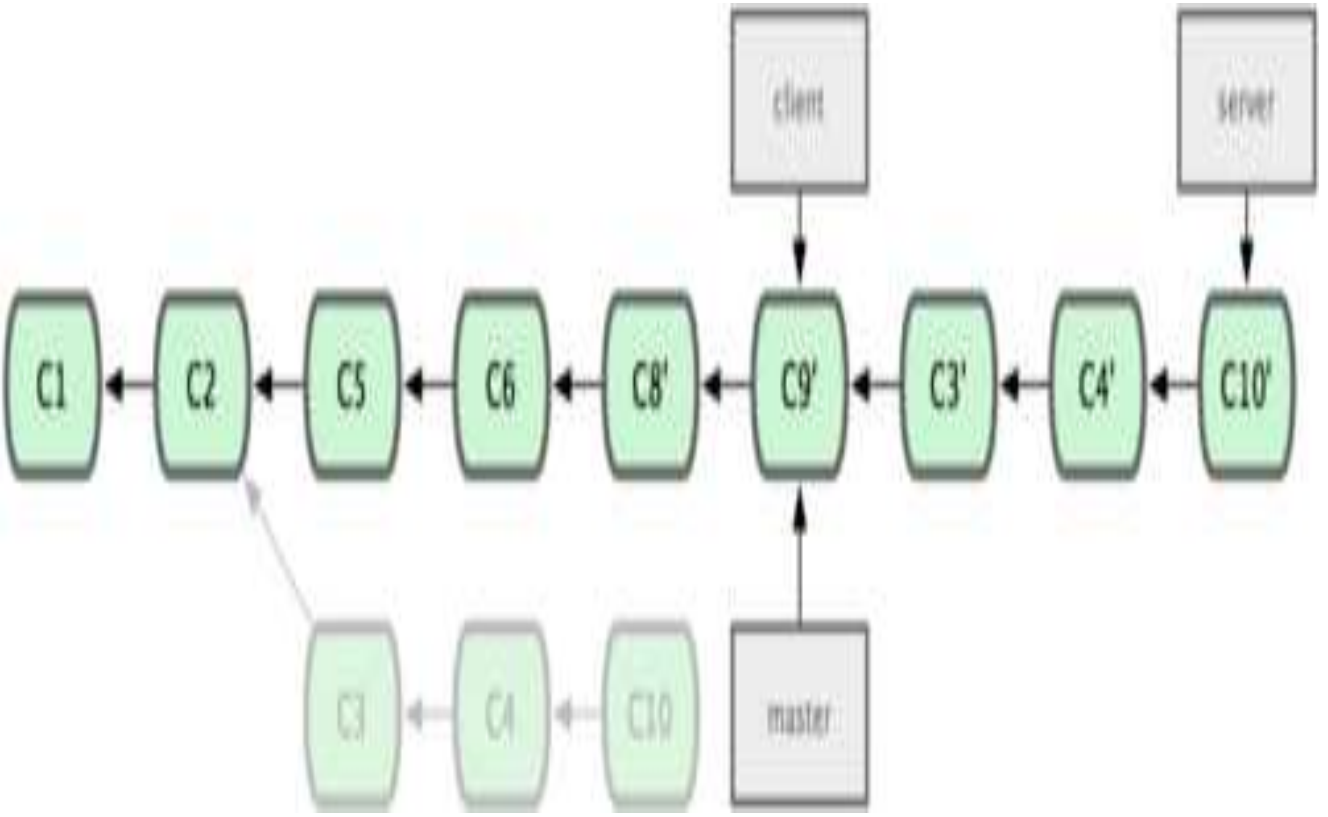


Figura 3-34. Reorganizando la rama server sobre la rama branch.

Después, puedes avanzar rápidamente la rama base (master):

```
$ git checkout master
$ git merge server
```

Y por último puedes eliminar las ramas client y server porque ya todo su contenido ha sido integrado y no las vas

a necesitar más. Dejando tu registro tras todo este proceso tal y como se muestra en la Figura 3-35:

```
$ git branch -d client
$ git branch -d server
```

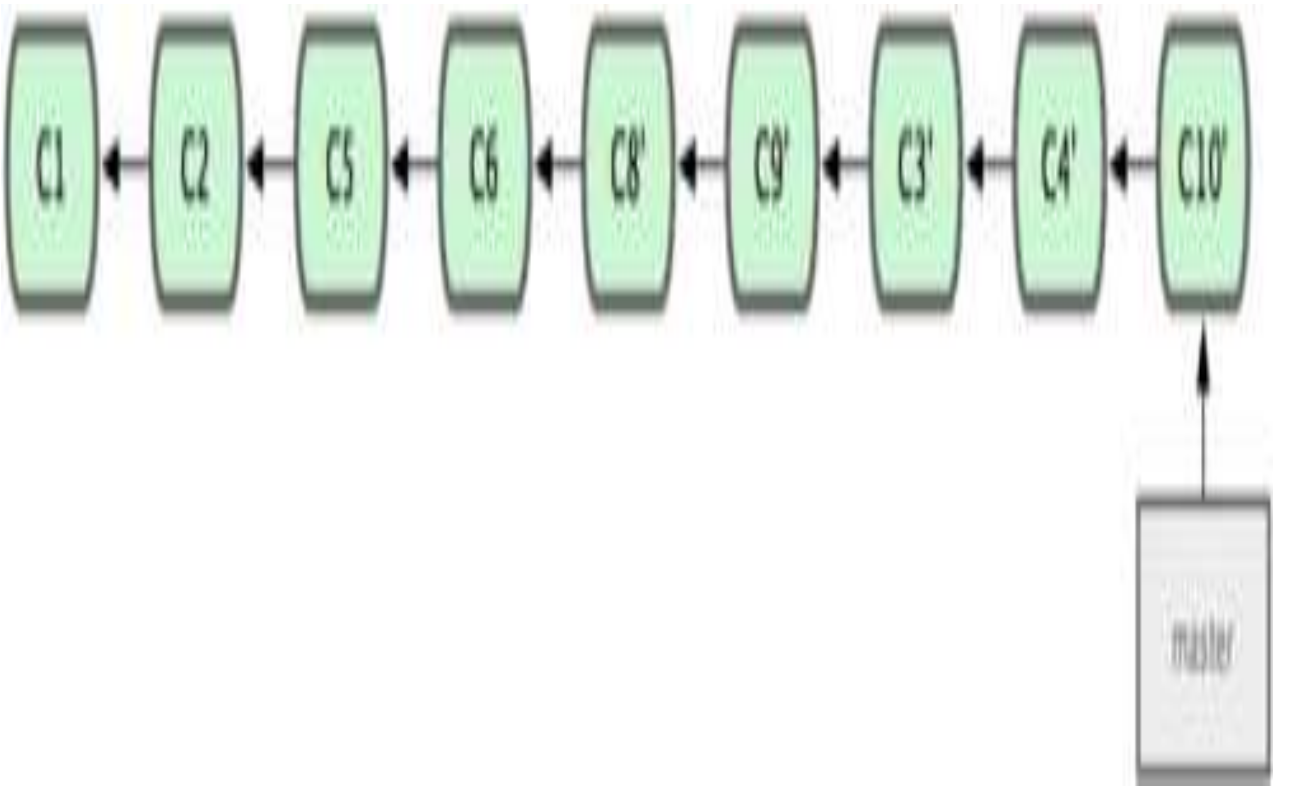


Figura 3-35. Registro final de confirmaciones de cambio.

Los peligros de la reorganización

Ahh..., pero la dicha de la reorganización no la alcanzamos sin sus contrapartidas:

Nunca reorganices confirmaciones de cambio (commits) que hayas enviado (push) a un repositorio público.

Siguiendo esta recomendación, no tendrás problemas. Pero si no la sigues, la gente te odiará y serás despreciado por tus familiares y amigos.

Cuando reorganizas algo, estás abandonando las confirmaciones de cambio ya creadas y estás creando unas nuevas; que son similares, pero diferentes. Si envias (push) confirmaciones (commits) a alguna parte, y otros las recogen (pull) de allí. Y después vas tu y las reescribes con `git rebase` **y las vuelves a enviar (push) de nuevo. Tus colaboradores tendrán que**

refusionar (re-merge) su trabajo y todo se volverá tremendamente complicado cuando intentes recoger (pull) su trabajo de vuelta sobre el tuyo.

Vamos a verlo con un ejemplo. Imagínate que haces un clon desde un servidor central, y luego trabajas sobre él. Tu registro de cambios puede ser algo como lo de la Figura 3-36.

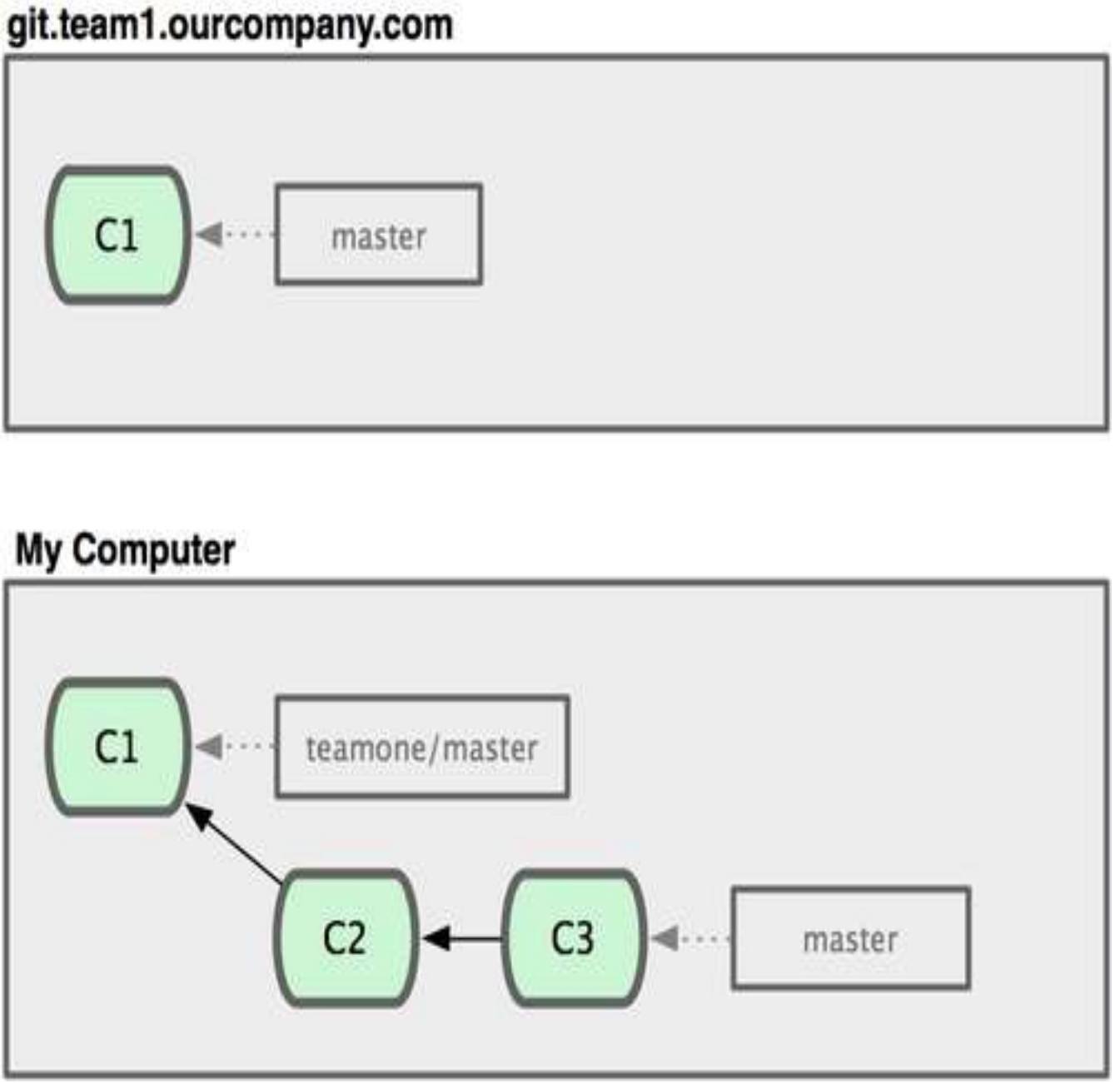


Figura 3-36. Clonar un repositorio y trabajar sobre él.

Ahora, otra persona trabaja también sobre ello, realiza una fusión (merge) y

lleva (push) su trabajo al servidor central. Tu te traes (fetch) sus trabajos y los fusionas (merge) sobre una nueva rama en tu trabajo. Quedando tu registro de confirmaciones como en la Figura 3-37.

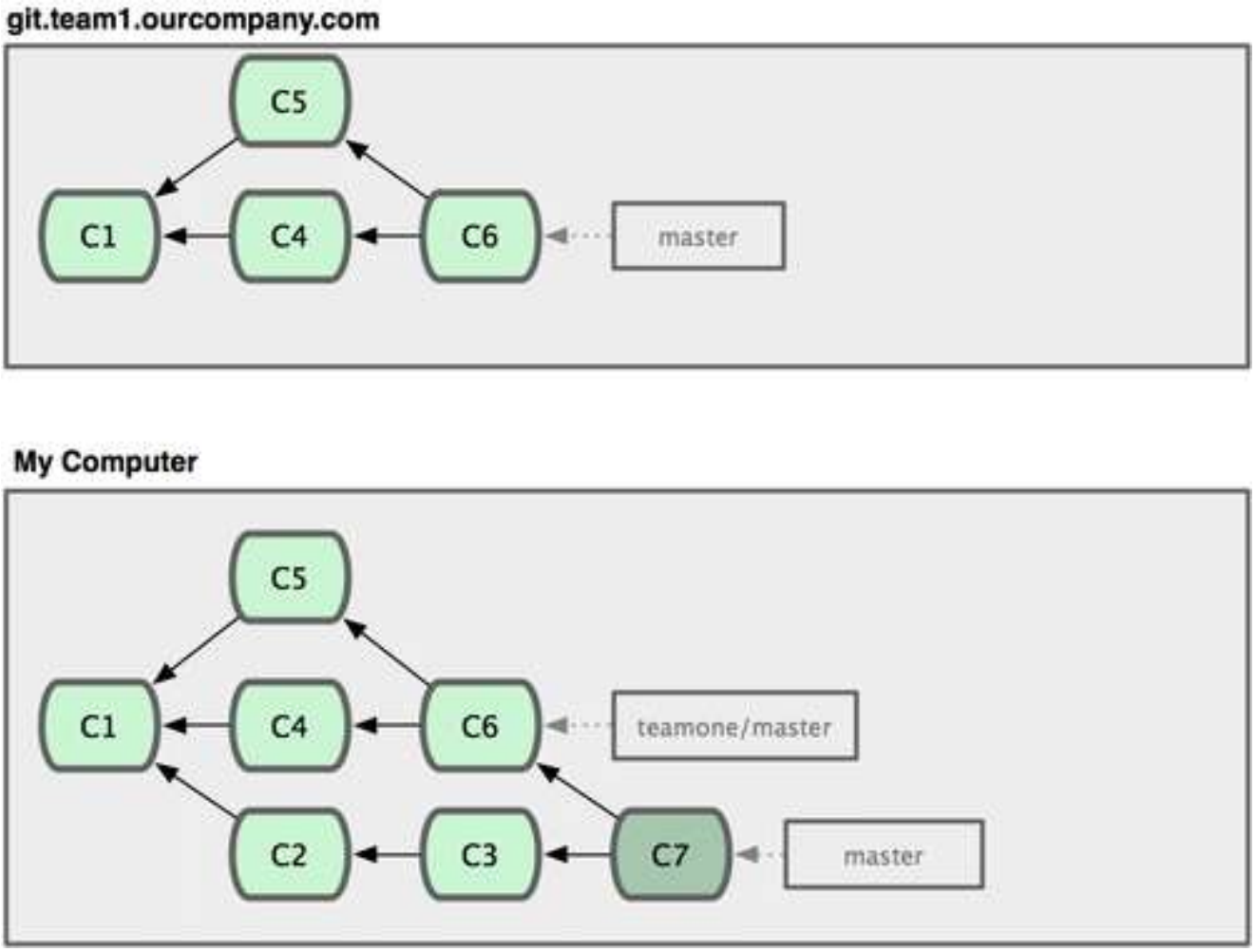


Figura 3-37. Traer (fetch) algunas

confirmaciones de cambio (commits) y fusionarlas (merge) sobre tu trabajo.

A continuación, la persona que había llevado cambios al servidor central decide retroceder y reorganizar su trabajo; haciendo un `git push --force` para sobrescribir el registro en el servidor. Tu te traes (fetch) esos nuevos cambios desde el servidor.

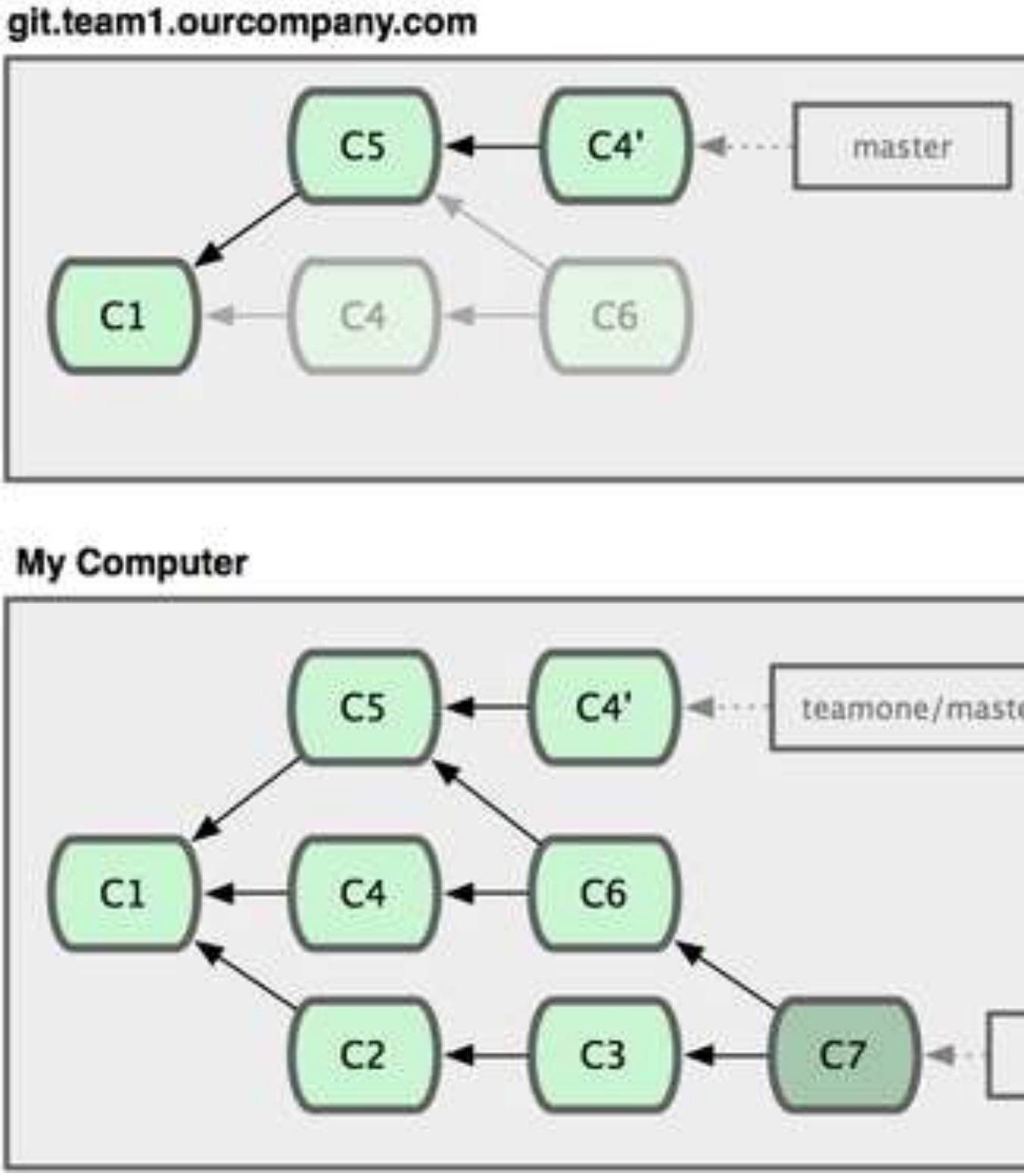


Figura 3-38. Alguien envií (push) confirmaciones (commits)

reorganizadas, abandonando las confirmaciones en las que tu habías basado tu trabajo.

En ese momento, tu te ves obligado a fusionar (merge) tu trabajo de nuevo, aunque creias que ya lo habias hecho antes. La reorganización cambia los resumenes (hash) SHA-1 de esas confirmaciones (commits), haciendo que Git se crea que son nuevas confirmaciones. Cuando realmente tu ya tenias el trabajo de C4 en tu registro.

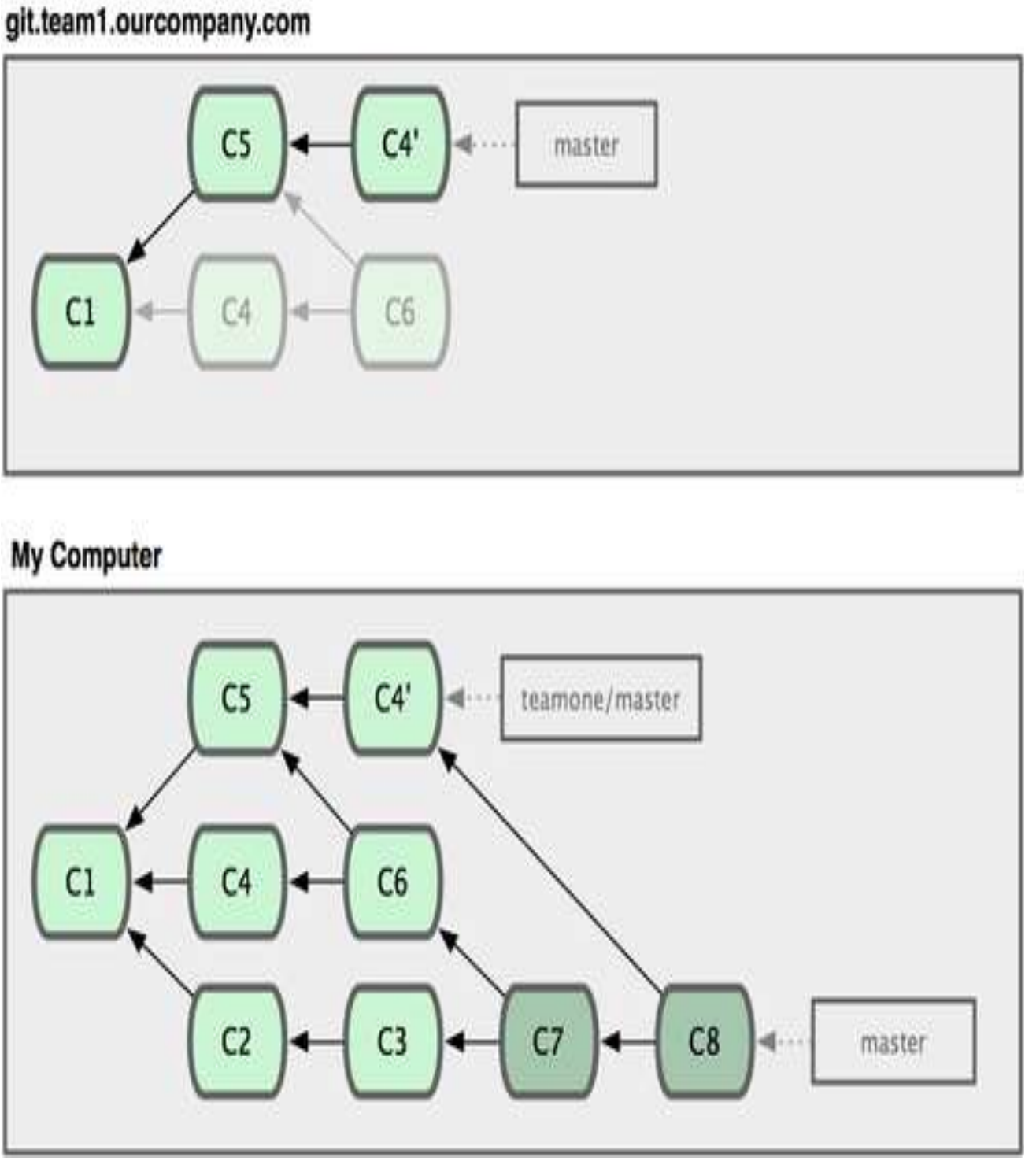


Figura 3-39. Vuelves a fusionar el mismo trabajo en una nueva fusión confirmada.

Te ves obligado a fusionar (merge) ese trabajo en algún punto, para poder seguir adelante con otros desarrollos en el futuro. Tras todo esto, tu registro de confirmaciones de cambio (commit history) contendrá tanto la confirmación C4 como la C4'; teniendo ambas el mismo contenido y el mismo mensaje de confirmación. Si lanzas un `git log` en un registro como este, verás dos confirmaciones con el mismo autor, misma fecha y mismo mensaje. Lo que puede llevar a confusiones. Es más, si luego tu envías (push) ese registro de vuelta al servidor, vas a introducir todas esas confirmaciones reorganizadas en el servidor central. Lo que puede confundir aún más a la gente.

Si solo usas la reorganización como una vía para hacer limpieza y organizar confirmaciones de cambio antes de enviarlas, y si únicamente reorganizas confirmaciones que nunca han sido

públicas. Entonces no tendrás problemas. Si, por el contrario, reorganizas confirmaciones que alguna vez han sido públicas y otra gente ha basado su trabajo en ellas. Entonces estarás en un aprieto.

7 Recapitulación

Hemos visto los procedimientos básicos de ramificación (branching) y fusión (merging) en Git. A estas alturas, te sentirás cómodo creando nuevas ramas (branch), saltando (checkout) entre ramas para trabajar y fusionando (merge) ramas entre ellas. También conocerás cómo compartir tus ramas enviándolas (push) a un servidor compartido, cómo trabajar colaborativamente en ramas compartidas, y cómo reorganizar (rebase) tus ramas antes de compartirlas.