Seminario 10: Git

Sistemas Distribuidos

Sara Balderas Díaz

Departamento de Ingeniería Informática
Universidad de Cádiz

Índice

Sistema de Control de Versiones (SCV) (I)

¿Qué es?

- Gestionar ficheros y versiones.
- Mecanismo para compartir ficheros.

¿Qué nos permite?

- Crear copias de seguridad.
- Sincronizar archivos.
- Deshacer cambios y/o restaurar versiones.
- Controlar la autoría del código.
- Realizar pruebas.

Sistema de Control de Versiones (SCV) (II)

Elementos básicos

- **Repositorio**: almacén que contiene el proyecto.
- **Servidor**: donde se aloja el repositorio.
- **Copias de trabajo locales**.
- **Rama**: localización dentro de un repositorio. La rama por defecto y principal del desarrollo se denominada habitualmente ***master***.

Sistema de Control de Versiones (SCV) (III)

Operaciones básicas

- **Add**: añade un archivo para que sea rastreado por el SCV.
- **Revisión**: versión de un archivo/directorio dentro del SCV.
- **Head**: última versión del repositorio (completo o de una rama).
- **Check out/clone**: crea una copia de trabajo que rastrea un repositorio.
- **Check in/commits**: envía los cambios locales al repositorio y cambia la versión del archivo(s)/repositorio. Puede tener asociado un mensaje descriptivo del cambio realizado.
- **Log**: histórico de cambios de un archivo/repositorio.
- **Update/Syncronize/fetch&pull**: sincroniza la copia de trabajo con la última versión que existe en el repositorio.
- **Revert/Reset**: deshace cambios realizados en la copia de trabajo cogiendo el último estado conocido del repositorio.

Sistema de Control de Versiones (SCV) (IV)

Operaciones avanzadas

- **Branches** (ramas):
  - Crea una copia de un recurso (archivo/carpeta) y se utilizan para trabajar con diferentes versiones de un repositorio al mismo tiempo.
  - Se pueden crear ramas adicionales para trabajar determinados aspectos y posteriormente unir el trabajo realizado.
- **Diff/change/Delta** (cambios): permite encontrar las diferencias entre dos versiones del repositorio.
- **Merge/Patch** (unir/fusionar): se utiliza habitualmente para unir/fusionar ramas.
- **Conflict** (conflicto): cambios solapados a causa de que se modifica el mismo recurso.

Sistema de Control de Versiones (SCV) (V)

SCV centralizados	SCV distribuidos
- Servidor centralizado que almacena el repositorio completo - La colaboración entre los participantes se hace a través del repositorio centralizado - Presentan ciertas restricciones - Son fáciles de usar - Herramientas: Subversion (SVN), Concurrent Version System (CVS), Microsoft Visual Source Safe, Perforce, etc.	- Cada participante posee una copia completa de todo el repositorio - La colaboración entre los participantes es más flexible - Su uso es más complejo que el de los SCV centralizados - Herramientas: Git, Mercurial, Bazaar, etc.

Git (I)

¿Qué es Git?

- Es un sistema distribuido de control de versiones (Version Control System, VCS) para el seguimiento de los cambios que se producen en archivos almacenados.
- No es necesario conocer ningún lenguaje de programación o framework para poder usar Git
- Gestionar una amplia variedad de proyectos (e.j., webs HTML estáticas, NodeJS, apps, Python, Java, etc.) de diferente tamaño, con rapidez y eficiencia.
- Línea de comandos y clientes gráficos tales como Tortoise, Giggle, etc.
- Es gratuito, de código abierto y rápido comparado con otras herramientas.
- Garantiza la integridad a partir de la suma de comprobación (checksummed) antes de ser almacenado.

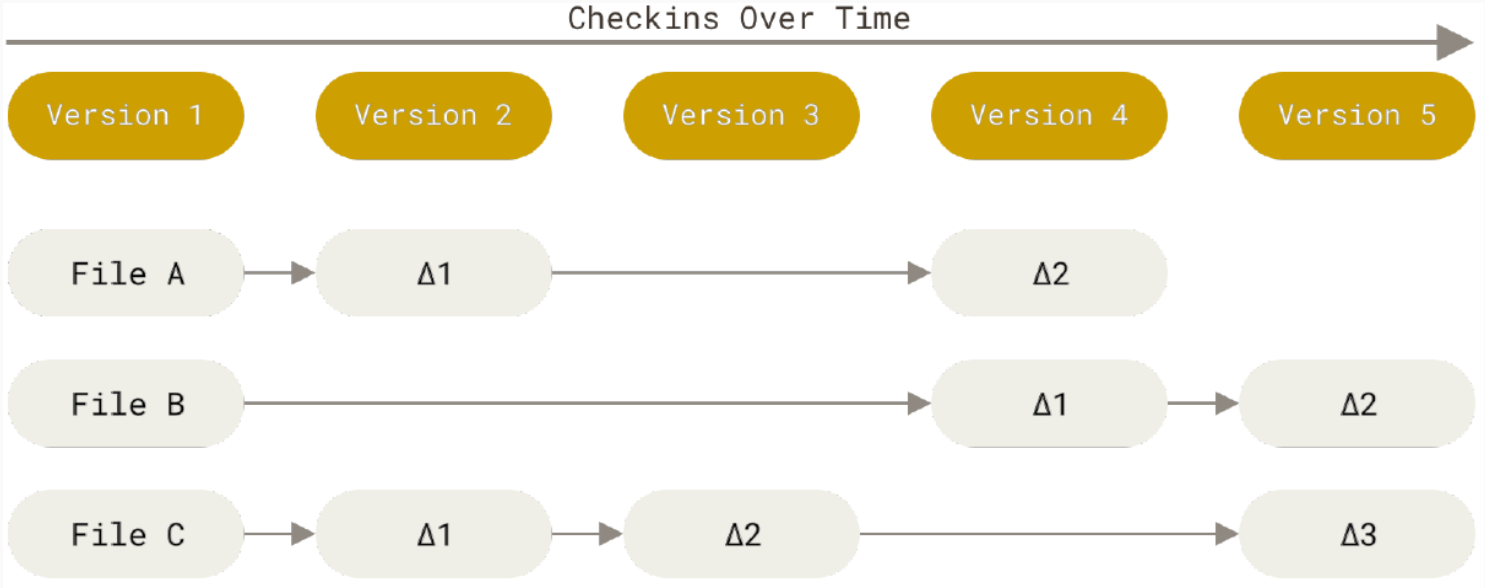
Git (II)

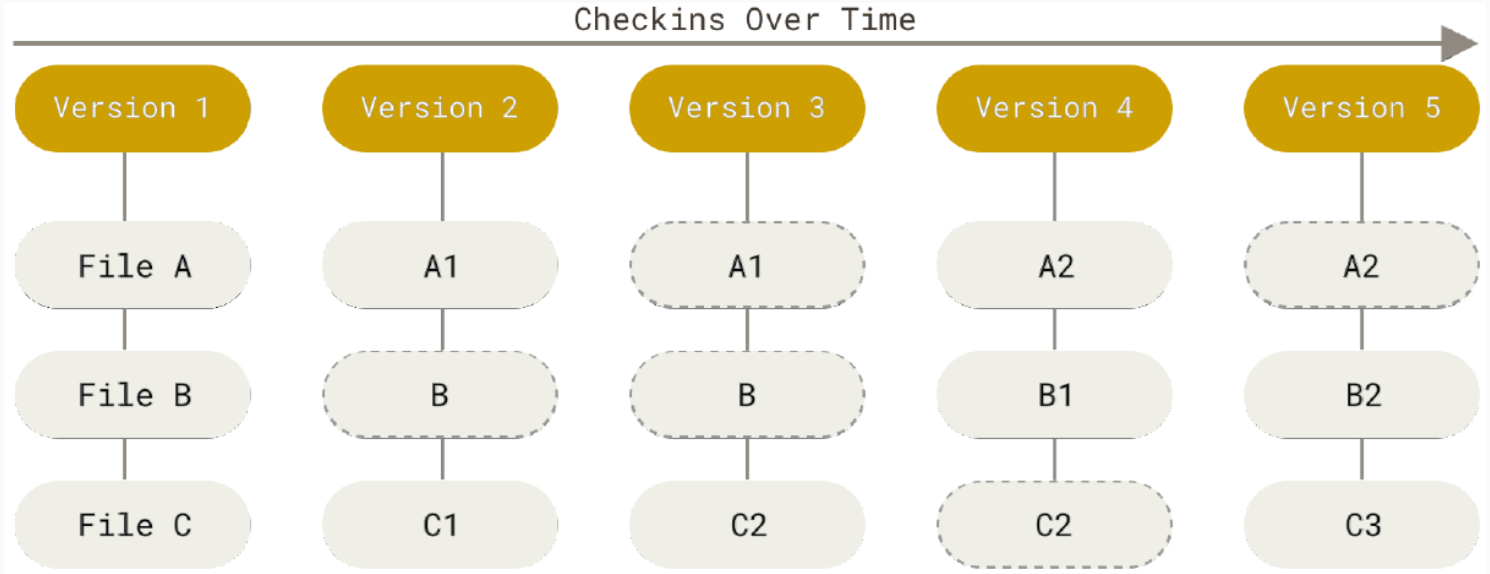
¿Qué ofrece Git?

- Desarrolladores pueden trabajar en un proyecto sin estar en la misma red.
- Coordinar el trabajo entre múltiples desarrolladores.
- Controlar quién hizo qué cambios, cuándo e incluso porqué.
- Recuperar versiones de cualquier archivo y momento siempre que se haya guardado en el repositorio.
- Repositorio local y remoto.
- Guardar una copia de todos los estados anteriores, modificaciones realizadas por los participantes, con comentarios y notas asociadas a cada cambio.
- Gestionar conflictos entre versiones.
- Gestionar diferentes ramas de proyecto.

Git VS otros SCV

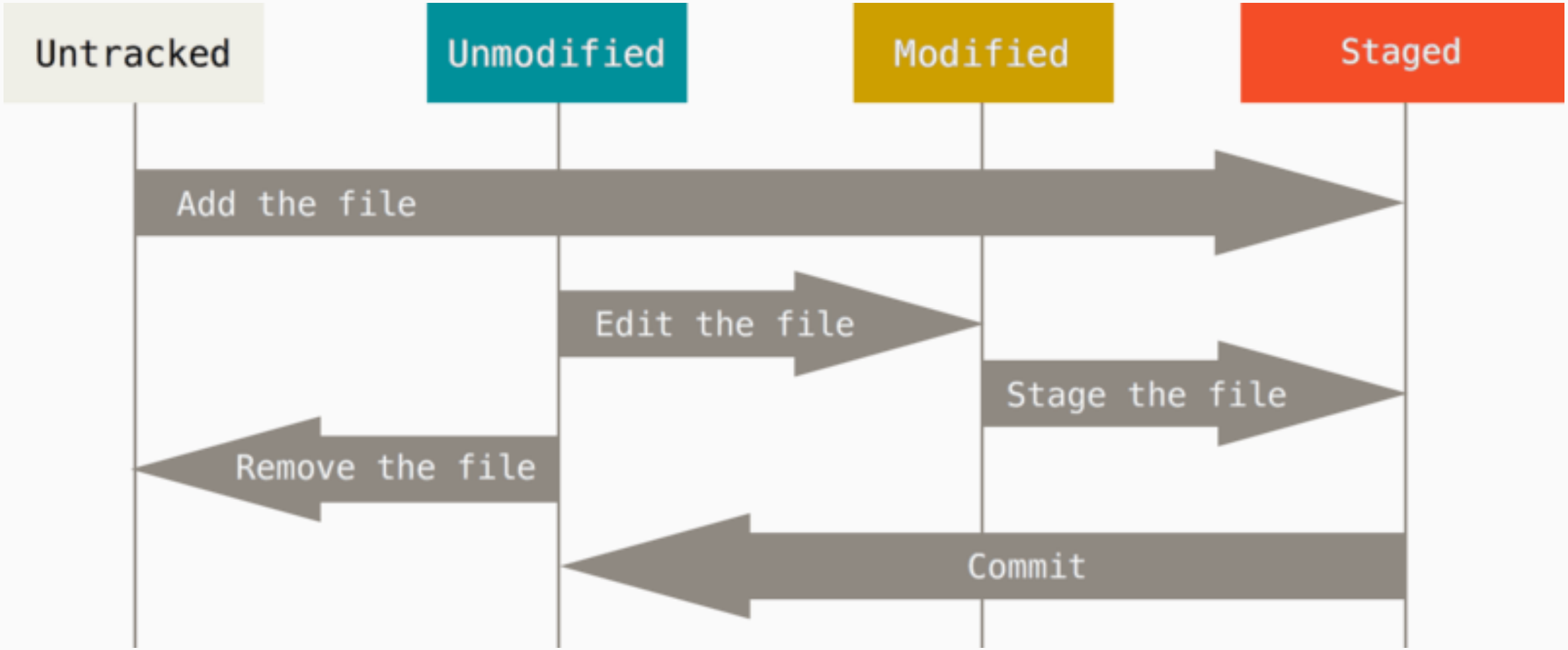
Otros SCV:
Almacenan los cambios sobre una versión base de cada fichero



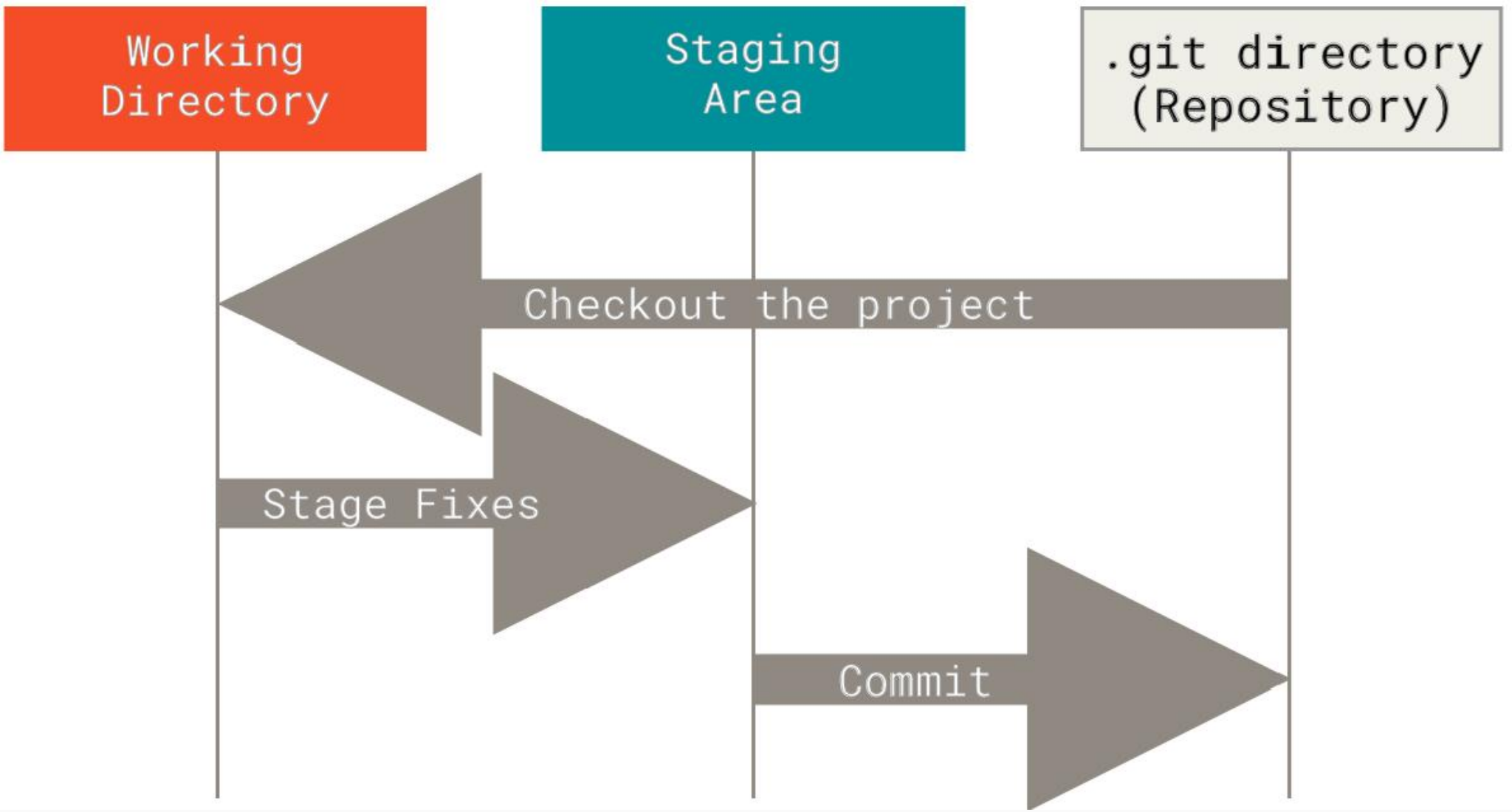


Git:
Sistemas de archivos en miniatura, captura una imagen de cómo están los archivos en ese momento y almacena la referencia a esa captura

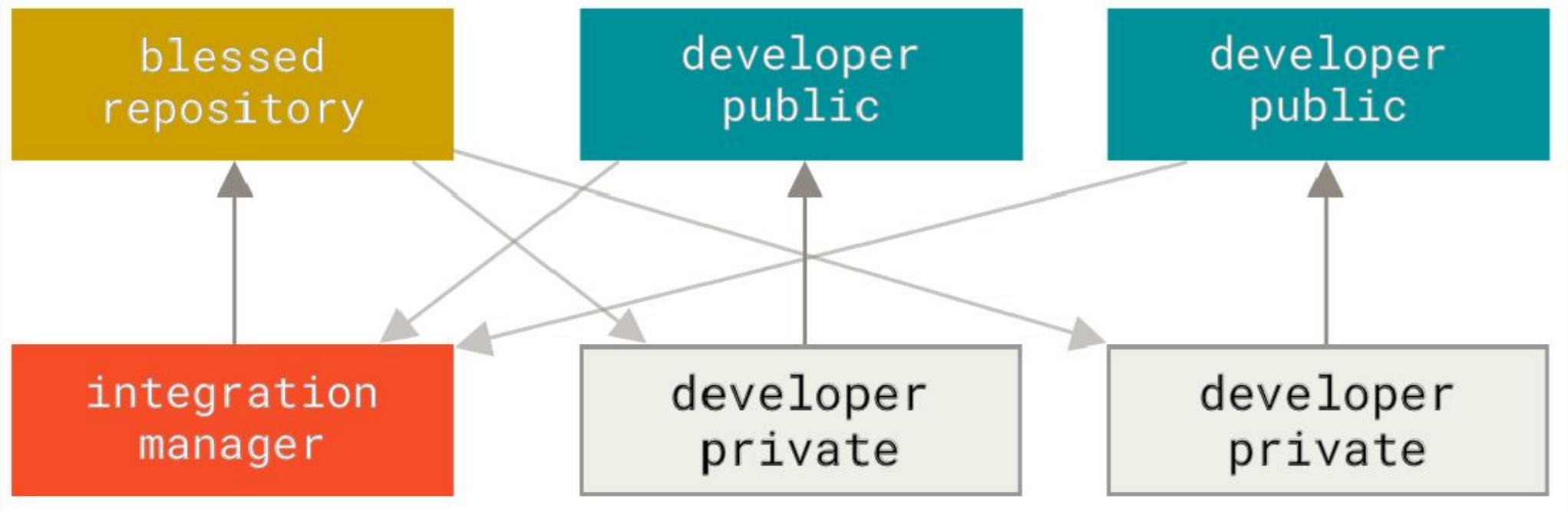
Estado de los archivos



Secciones


Working Directory
Staging Area
.git directory (Repository)
Checkout the project
Stage Fixes
Commit

Flujos de trabajo



Metodología de trabajo

1. Opciones:
    - crear un repositorio del proyecto y crear copia de trabajo del repositorio o;
    - crear proyecto en local e importar inicialmente el código del proyecto.
1. Realizar cambios.
2. Enviar cambios al repositorio.

En posteriores ocasiones, antes de hacer nada, debemos actualizar el repositorio.

Comandos Básicos

Repositorio Local	Repositorio Remoto
// Inicializar el repositorio Local de Git `$ git init`	// Enviar cambios al repositorio remoto `$ git push <nombre_remoto> <nombre_rama>`
// Registrar cambios de archivo(s) (añade al Index) `$ git add <archivo>`	// Descargar los datos del repositorio remoto y actualizar (integrar y fusionar) el repositorio local `$ git pull`
// Registrar cambios de todos los archivos (añade al Index) `$ git add .`	// Clona un repositorio en un nuevo directorio, crea ramas de seguimiento remoto para cada rama en el repositorio clonado, y crea y extrae una rama inicial que se bifurca desde la rama actualmente activa del repositorio clonado `$ git clone`
// Mostrar el estado del repositorio `$ git status`	
// Guardar cambios en el repositorio local `$ git commit`	
// Guardar cambios en el repositorio local con un comentario `$ git commit –m “Commit message”`	

Otros comandos de interés

// Ayuda sobre los comandos de git

```
$ git help <comando>
```

// Muestra los archivos modificados pero no añadidos al staging area

```
$ git diff
```

// Elimina el archivo

```
$ git rm <archivo>
```

// Renombrar (mover) archivo

```
$ git mv <origen> <destino>
```

// Visualizar la historia de los commits

```
$ git loop
```

// Mostrar el autor que ha modificado por última vez cada línea de un archivo

```
$ git blame <fichero>
```

Ignorando Archivos (I)

El archivo **.gitignore** nos permite especificar qué archivos o conjunto de archivos no queremos rastrear con Git.

De esta forma, podemos evitar tener en el repositorio archivos innecesarios, tales como ejecutables, logs, archivos temporales, etc.

Aquí podéis encontrar varias plantillas de archivos .gitignore para diferentes lenguajes de programación: https://github.com/github/gitignore

Ignorando Archivos (II)

Si existe una **carpeta vacía** en nuestro proyecto, Git no la creará en el repositorio.

Existen casos en los que necesitamos esas carpetas vacías, para ello, podemos crear dentro de la carpeta un fichero vacío cualquiera.

Por convención, este fichero se suele llamar **.gitkeep**

Ignorando Archivos (III)

Además de especificar los nombres de los archivos que queremos que Git ignore, podemos hacer uso de patrones:

```
#ignorar todos los archivos ejecutables

*.exe


#ignorar los archivos de un directorio (ignorará también la carpeta)

temp/


#ignorar todos los archivos .html excepto los incluidos en la carpeta main

*.html

!main/*.html
```

Tagging (I)

El uso de **tags** nos permite marcar los hitos importantes en el desarrollo de un proyecto.

El tag marca al proyecto completo, no a un archivo particular, en su estado actual.

Suelen utilizarse para versionar el proyecto (p. ej., v1.0.2).

Los tag son inmutables, no pueden cambiar una vez creados (sí eliminarse, aunque no es recomendable hacerlo).

Tagging (II)

Para crear nuestro primer tag, hacemos:

```
$ git tag v0.0.1 -m “Primera versión”
```

Esto asignará al proyecto el tag v0.0.1 cuando hagamos nuestro siguiente commit.

Posteriormente al commit, podemos asignar un nuevo tag:

```
$ git tag v0.0.2 -m “Cambios menores”
```

Tagging (III)

Si queremos consultar los tags de nuestro proyecto:

```
$ git tag
v0.0.1
v0.0.2
```

También podemos ver información acerca del autor del tag, fecha de creación y estado del repositorio con un tag concreto:

```
$ git show v0.0.1
```

No hay que olvidar enviar los tag creados localmente al repositorio externo:

```
$ git push --tags
```

Inicio con Git

Descargar e instalar Git:

- Versiones disponibles para Windows, Linux/Unix y Mac OS X en https://git-scm.com/downloads
- Terminal:
  - Linux (Debian): `$ sudo apt-get install git`
  - Linux (Fedora): `$ sudo yum install git`

Documentación:

Manuales, guías, vídeos, comandos, etc. disponibles en https://git-scm.com/doc

Primeros pasos

1) Lanzar Git.

2) Configurar *nombre de usuario* y *email*:

```
$ git config --global user.name ‘Sara Balderas’

$ git config --global user.email ‘sara.balderas@uca.es’
```

3) Creamos un proyecto.

4) Añadimos el archivo al repositorio git.

```
$ git add clientUDP.py
```

5) Comprobamos el estado.

```
$ git status
```

Repositorio

- Crear un nuevo repositorio desde la línea de comandos

```
echo "# prueba" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin
https://github.com/sarabalderasdiaz/prueba.git
git push -u origin master
```

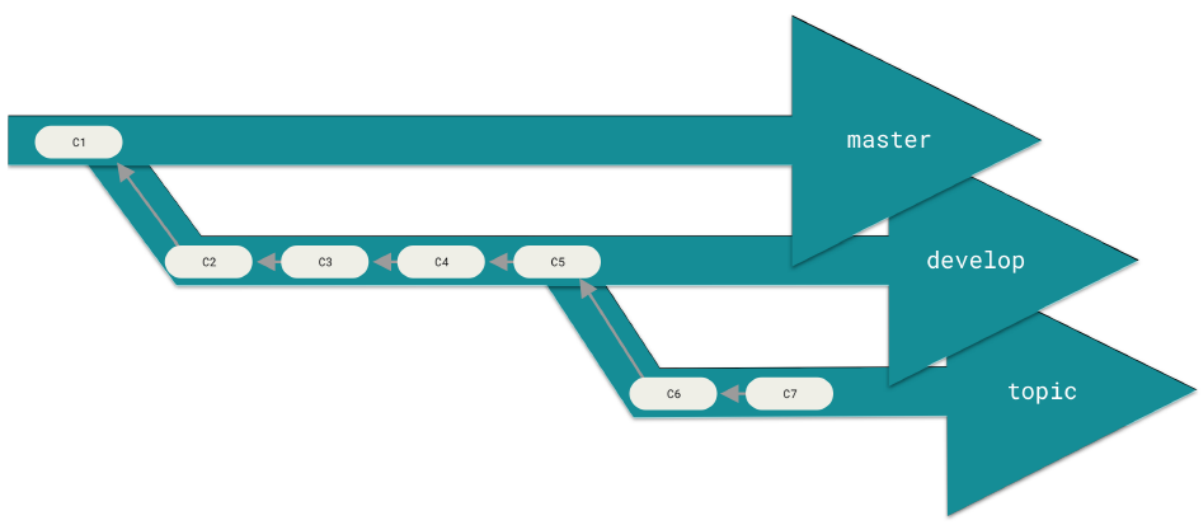
- Push un repositorio existente desde la línea de comandos

```
git remote add origin https://github.com/sarabalderasdiaz/prueba.git
git push -u origin master
```

- Importar código desde otro repositorio (Subversion, Mercurial, o TFS)

Branches (ramas) (I)

- Son fáciles de crear y borrar
- Pueden ser públicas y privadas
- Facilitan la organización del trabajo y los experimentos



Comandos

// Crear una rama (local) a partir de la rana actual

```
$ git branch <nombre_rama>
```

// Cambiar de rama de trabajo

```
$ git checkout <nombre_rama>
```

// Crear y cambiar de rama de trabajo (al mismo tiempo)

```
$ git checkout -b <nombre_rama>
```

// Unir ramas

```
$ git merge <nombre_rama>
```

// Ver el último commit en cada rama

```
$ git branch -v
```

// Filtrar la lista de ramas que hemos unido y las que no

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

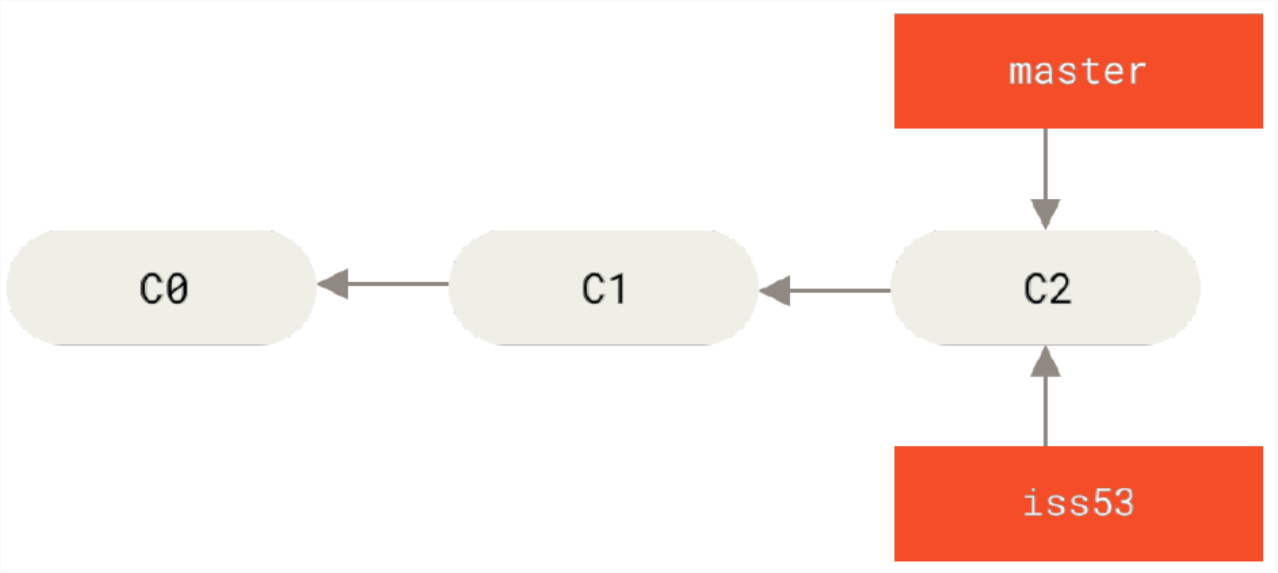
Branches (ramas) (II)

Opción 1:

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

Opción 2:

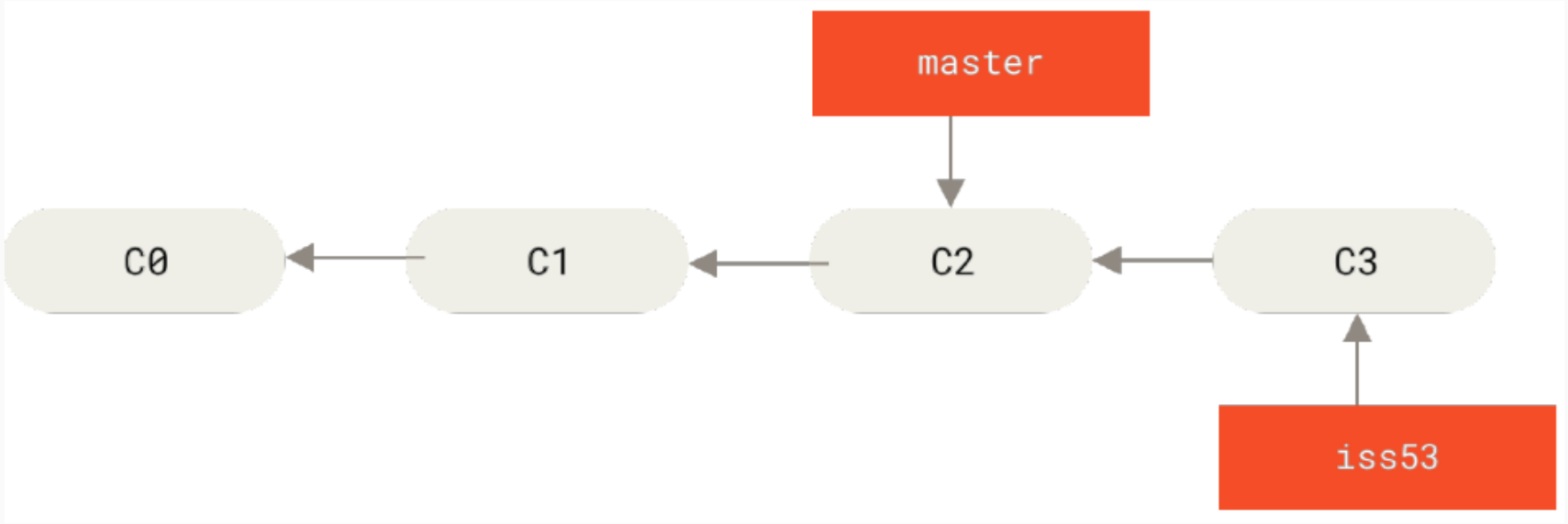
```
$ git branch iss53
$ git checkout iss53
```



Branches (ramas) (III)

Hacemos un cambio en nuestro proyecto y:

```
$ git commit -a -m 'Change IP'
```

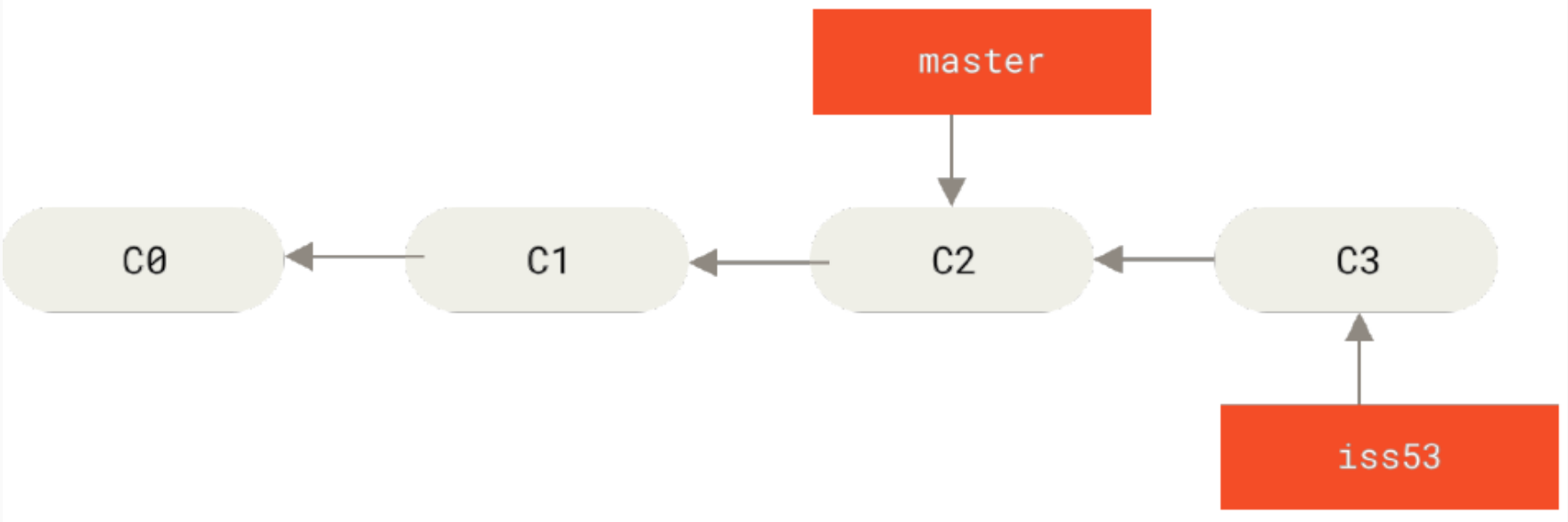


Branches (ramas) (III)

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

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
```

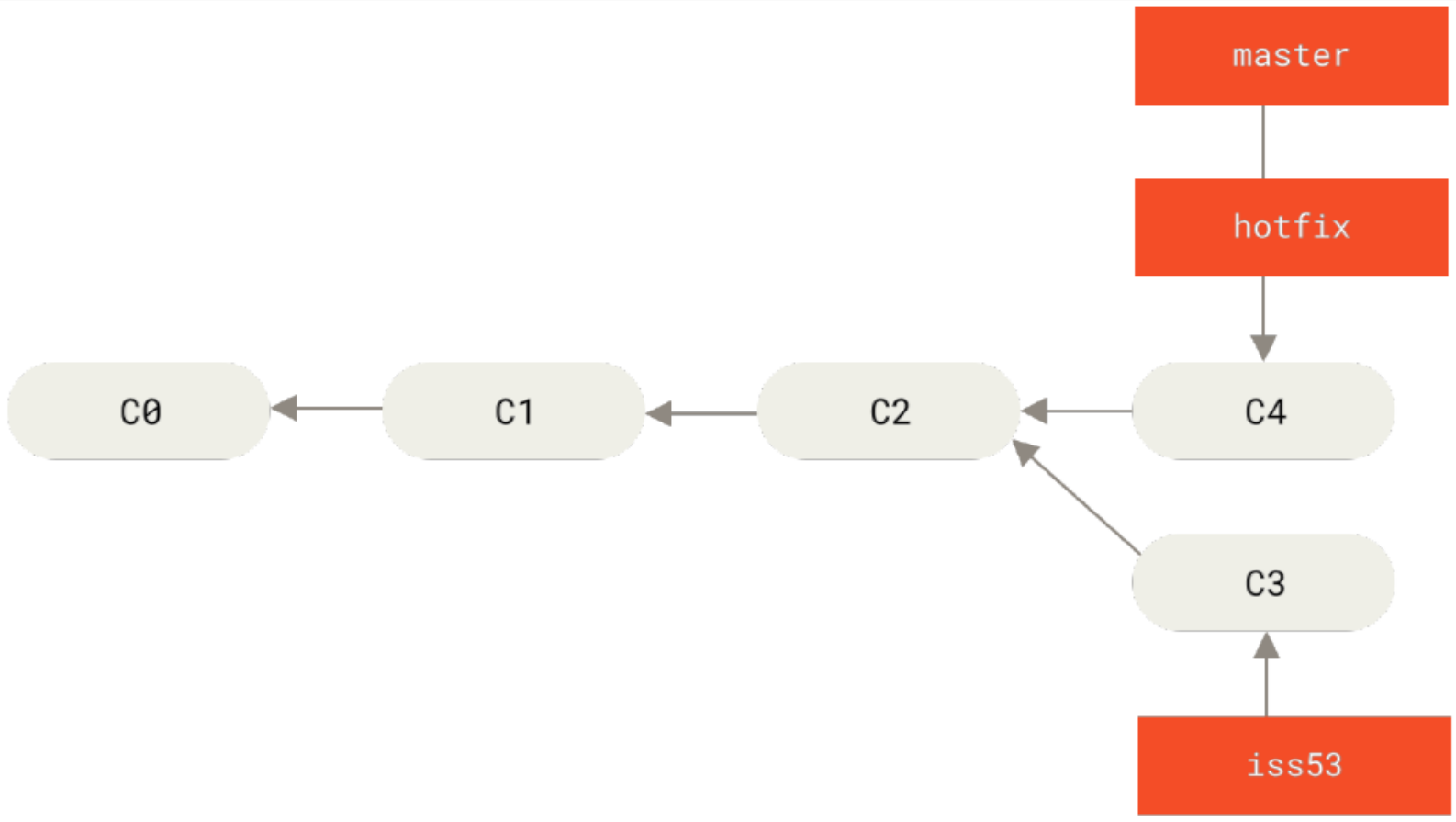
```
$ git commit -a -m 'Fix URL'
[hotfix 1fb7853] Fix URL
1 file changed, 2 insertions(+)
```



Branches (ramas) (IV)

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

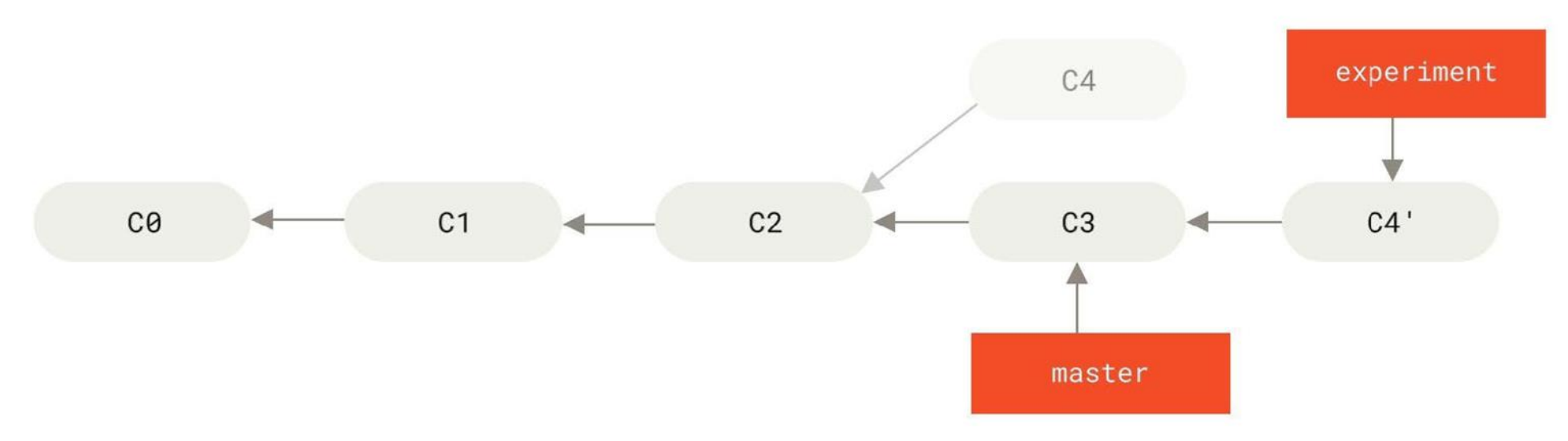
$ git merge hotfix
Updating r32b972..4j7492t
Fast-forward
clientUDP.py | 2 ++
1 file changed, 2 insertions(+)



Branches (ramas) (V)

```
$ git checkout experiment
$ git rebase master
```

First, rewinding head to replay your work on top of it...
Applying: added staged command



Bibliografía

- https://git-scm.com/

- Chacon, S., & Straub, B. (2014). Pro Git. Apress. https://git-scm.com/book/en/v2

- Hodson, R. (2014). Ry's Git Tutorial.