Contenido

[**¿QUÉ ES GIT?** 2](#_Toc21337291)

[**Sistemas de Control de Versiones Distribuidos** 2](#_Toc21337292)

[**Funcionamiento de las réplicas** 3](#_Toc21337293)

[**SISTEMAS DE CONTROL DE VERSIONES CENTRALIZADOS VS SISTEMAS DE CONTROL DE VERSIONES DISTRIBUIDOS** 3](#_Toc21337294)

[**DIFERENTES ESTADOS DE UN FICHERO GIT** 4](#_Toc21337295)

[**Secciones principales de un proyecto Git** 4](#_Toc21337296)

[**ORDENES BÁSICAS** 4](#_Toc21337297)

[**BUENAS PRÁCTICAS** 5](#_Toc21337298)

# **¿QUÉ ES GIT?**

Git es un sistema de control específico de versión de fuente abierta creada por Linus Torvalds en el 2005.

Específicamente, Git es un sistema de control de versión distribuida, lo que quiere decir que la base del código entero y su historial se encuentran disponibles en la computadora de todo desarrollador, lo cual permite un fácil acceso a las bifurcaciones y fusiones. Según la encuesta entre los desarrolladores de Stack Overflow, más del 87% de los desarrolladores usan Git.

Git es una herramienta que realiza una función del control de versiones de código de forma distribuida, de la que destacamos varias características:

* Es muy potente
* Fue diseñada por Linus Torvalds
* No depende de un repositorio central
* Es software libre
* Con ella podemos mantener un historial completo de versiones
* Podemos movernos, como si tuviéramos un puntero en el tiempo, por todas las revisiones de código y desplazarnos una manera muy ágil.
* Es muy rápida
* Tiene un sistema de trabajo con ramas que lo hace especialmente potente
* En cuanto a la funcionalidad de las ramas, las mismas están destinadas a provocar proyectos divergentes de un proyecto principal, para hacer experimentos o para probar nuevas funcionalidades.
* Las ramas pueden tener una línea de progreso diferente de la rama principal donde está el core de nuestro desarrollo. En algún momento podemos llegar a probar algunas de esas mejoras o cambios en el código y hacer una fusión a nuestro proyecto principal, ya que todo esto lo maneja Git de una forma muy eficiente

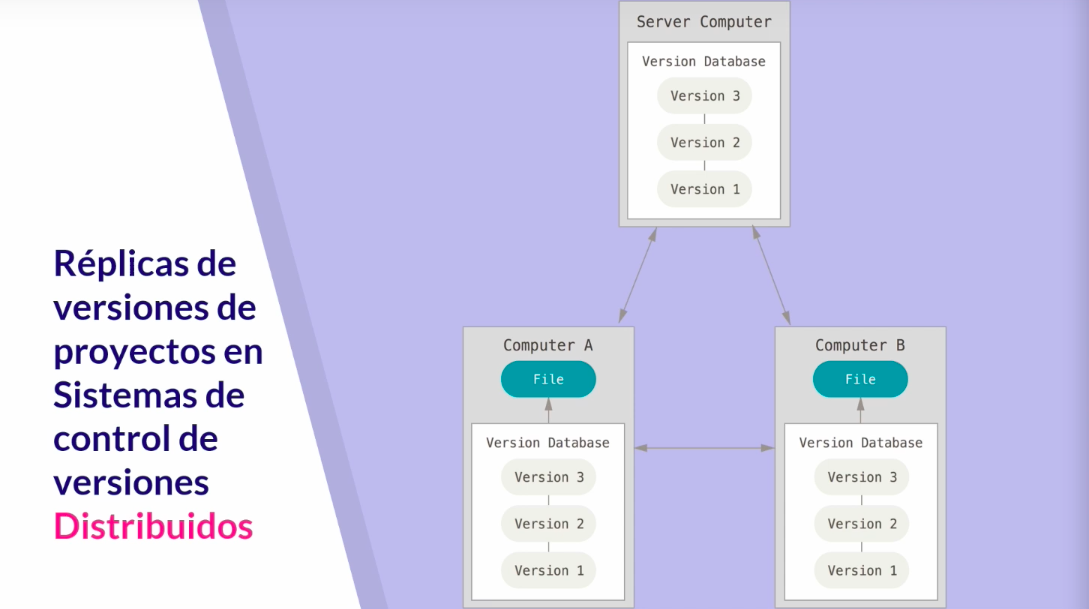
## **Sistemas de Control de Versiones Distribuidos**

Los Sistemas de Control de Versiones Centralizados (VCS), como por ejemplo Subversion, que es una herramienta en la que se ha confiado para albergar el histórico de revisión de versiones, es un punto centralizado, lo cual puede llegar a suponer una merma de trabajo si perdemos la conectividad de la red.

Los Sistemas de Control de Versiones Distribuidos (DVCS) salvan este problema. Algunos ejemplos de sistemas distribuidos, aparte de Git, son Mercurial, Bazaar o Darcs. En este tipo de herramientas, los clientes replican completamente el repositorio.

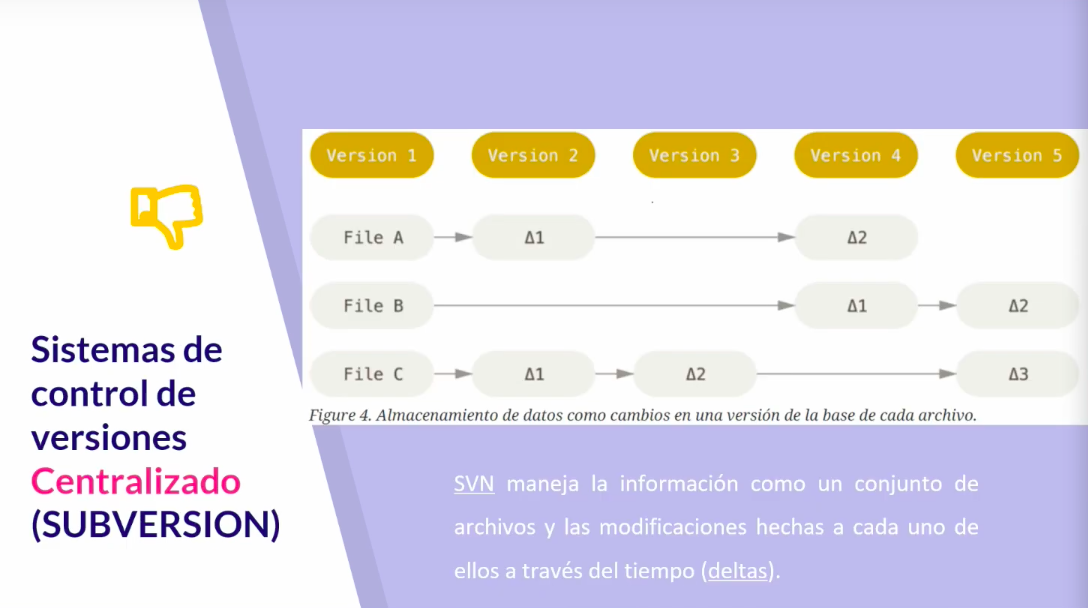
## **Funcionamiento de las réplicas**

Las réplicas de versiones de proyectos en Sistemas de Control de Versiones Distribuidos funcionan como podemos ver en este gráfico:



Podemos tener una máquina que actúa como servidor, en el mismo se van almacenando las diferentes versiones de nuestro código, y cada uno de los clientes que participen en ese desarrollo tiene el histórico de las revisiones completa.

## **SISTEMAS DE CONTROL DE VERSIONES CENTRALIZADOS VS SISTEMAS DE CONTROL DE VERSIONES DISTRIBUIDOS**

****

En Sistemas de Control de Versiones Centralizados, como Subversion, partimos de una versión, por ejemplo de la versión 1 y tenemos tres ficheros, A, B y C.

De la versión 1 a la versión 2, las diferencias, como si fueran esa especie de incrementos que llamamos deltas, son almacenados por el sistema. De esta manera, por buscar algún símil, es como si Subversion trabajase con backups incrementales.

Después tenemos el funcionamiento de Sistemas de Control de Versiones de Git, en el que cada vez que hay cambios de ficheros éste es almacenado otra vez, y si no hay cambios es como si tuviésemos una especie de apuntador al fichero que no ha tenido cambios, en una revisión o en un hito del tiempo anterior.

# **DIFERENTES ESTADOS DE UN FICHERO GIT**

Existen tres tipos de estado de un fichero Git:

## **Secciones principales de un proyecto Git**

* Confirmado: tenemos un fichero con el que hemos estado trabajando, hemos aprobado todos sus cambios y va en una nueva revisión, es decir, una especie de paquete dónde van todos los cambios.
* Modificado: estamos trabajando en el directorio de trabajo (o working directory) y consideramos que ese cambio debe de ir en una revisión, en esa especie de paquete, para formar la revisión.
* reparado: hemos marcado un archivo para que vaya una revisión.
* El área del working directory, que es dónde vamos a tener todos nuestros ficheros, dónde estamos trabajando constantemente.
* El staging area, que es donde van los archivos que estamos modificando y que aceptamos para que vayan en una futura revisión.
* El área de commit o el git directory, que es dónde se almacenan la revisión completa. A lo largo de nuestro curso de Git, se explicará cómo podemos movernos a lo largo de esos tres estados, para qué sirven y por qué suponen una ventaja.

## **ORDENES BÁSICAS**

1. Iniciar un repositorio vacío en una carpeta específica.
2. git init
3. Añadir un archivo específico.
4. git add “nombre\_de\_archivo”
5. Añadir todos los archivos del directorio
6. git add .
7. Confirmar los cambios realizados. El “mensaje” generalmente se usa para asociar al commit una breve descripción de los cambios realizados.
8. git commit –am “mensaje”
9. Revertir el commit identificado por "hash\_commit"
10. git revert “hash\_commit"
11. Subir la rama(branch) “nombre\_rama” al servidor remoto.
12. git push origin “nombre rama”
13. Mostrar el estado actual de la rama (branch), como los cambios que hay sin hacer commit.
14. git status

## **BUENAS PRÁCTICAS**

Cada desarrollador o equipo de desarrollo puede hacer uso de Git de la forma que le parezca más conveniente. Sin embargo, una buena práctica es la siguiente, utilizando 4 tipos de ramas: Master, Development, Features, y Hotfix.

* **Master**: Es la rama principal. Contiene el repositorio que se encuentra publicado en producción, por lo que debe estar siempre estable.
* **Development**: Es una rama sacada de Master. Es la rama de integración, todas las nuevas funcionalidades se deben integrar en esta rama. Luego que se realice la integración y se corrijan los errores (en caso de haber alguno), es decir que la rama se encuentre estable, se puede hacer un merge de development sobre la rama Master.
* **Features**: Cada nueva funcionalidad se debe realizar en una rama nueva, específica para esa funcionalidad. Estas se deben sacar de Development. Una vez que la funcionalidad esté desarrollada, se hace un merge de la rama sobre Development, donde se integrará con las demás funcionalidades.
* **Hotfix**: Son errores de software que surgen en producción, por lo que se deben arreglar y publicar de forma urgente. Es por ello, que son ramas sacadas de Master. Una vez corregido el error, se debe hacer una unificación de la rama sobre Master. Al final, para que no quede desactualizada, se debe realizar la unificación de Master sobre Development.