


**UNIVERSIDAD DE EL SALVADOR
FACULTAD DE INGENIERÍA Y ARQUITECTURA
ESCUELA DE INGENIERÍA**



**TECNOLOGÍA ORIENTADA A OBJETOS (TOO-115)
PROYECTO DE INVESTIGACIÓN TECNOLOGÍAS DE VANGUARDIA**

**TEMA:
SISTEMAS DE CONTROL DE VERSIONES:
CENTRALIZADOS Y DISTRIBUIDOS**

GRUPO 15:	
CGM15	

CARNET	NOMBRES
CM15044	CORTEZ MEJÍA, BERNARDO ANTONIO
GR11057	GÓMEZ RUIZ, JONATHAN ALEXANDER
GS02009	GONZÁLEZ SERRANO, NELSON ARNOLDO
MS15053	MENJÍVAR SÁNCHEZ, WILBERT ULISES
PQ15006	PERAZA GONZÁLEZ, ÁNGEL ALEXANDER

Tabla De Contenido

INTRODUCCIÓN	iii
OBJETIVOS	iv
MARCO TEÓRICO	1
1. Sistemas de Control de Versiones.	1
A. Definición.....	1
B. Características De Un Sistema de Control de Versiones	1
C. Tipos de Sistemas de Control de Versiones	2
i. Sistemas de Control de Versiones Centralizados.	2
a) Ventajas De Sistemas Centralizados.....	2
b) Desventajas De Sistemas Centralizados	2
ii. Sistemas de Control de Versiones Distribuidos.	3
a) Ventajas De Sistemas Distribuidos	3
b) Desventajas De Sistemas Distribuidos.....	4
D. EJEMPLOS DE SISTEMAS DE VERSIONES CENTRALIZADOS.....	4
a) CVS	4
b) SUBVERSIÓN	5
E. EJEMPLOS DE SISTEMAS DE VERSIONES DISTRIBUIDOS	5
a) Mercurial.....	5
b) GIT.....	6
F. FUNDAMENTOS DE GIT.....	6
G. SERVICIOS WEB DE CONTROL DE VERSIONES BASADOS EN GIT	8
a) GitHub	8
b) GitLab	8
H. DIFERENCIAS ENTRE GITHUB Y GITLAB.....	8
i. Colaboradores por repositorio	9
ii. Tamaño máximo.....	9
iii. Alternativas a GitHub gratuitas que puedes montar en tu servidor.....	9
iv. Diferencia de usuarios	9
v. GitLab: Alternativa favorita para la inminente compra de GitHub por Microsoft	9
I. Cuadro Comparativo Sistemas de Control de Versiones Centralizados y Distribuidos.....	10
J. CUANDO ELEGIR GIT Y CUANDO SUBVERSION.....	11
K. GUIA BÁSICA DE SUBVERSIÓN SVN (Cdmon., 2018).....	12



i.	Copiar el repositorio.....	12
ii.	Añadir fichero al repositorio	12
iii.	Algunos comandos de SVN	12
L.	GUÍA BÁSICA DE GIT. – COMANDOS. (Juan José Lozano Gómez, 2019).....	13
i.	Comandos Git para crear un repositorio local	13
ii.	Integrando un repositorio remoto	13
iii.	Comandos Git para confirmar los cambios: add y commit	13
iv.	Integrando los cambios en tu repositorio remoto	14
v.	Comandos Git para crear ramas	14
vi.	Fusionando el contenido de una rama en otra	14
vii.	Comandos Git para deshacer los cambios: revert	14
M.	EJEMPLO DE PROYECTO EN GIT USANDO GITHUB. (Productividad, 2019)	15
N.	EJEMPLO PASO A PASO DE SOLUCIÓN CONFLICTOS EN GIT USANDO GITHUB. 16	
O.	GUÍA BÁSICA DE MERCURIAL. (Quevedo, 2012).....	19
i.	Creación de repositorio	19
ii.	Nuevos archivos	19
iii.	Confirmación de cambios en un archivo.....	20
iv.	Creación de copia personal de repositorio.....	20
v.	Revisión del historial.....	21
vi.	Actualizaciones locales	21
vii.	Comparar repositorio local y maestro	22
viii.	Actualizar el repositorio maestro	22
ix.	Actualizar el repositorio local	23
CONCLUSIONES		24
BIBLIOGRAFÍA		25

INTRODUCCIÓN

El presente documento mostrará algunas de las herramientas de control de versiones, que, como sabemos, éstas herramientas están diseñadas para facilitar, ayudar, y mejorar el modo de programación para grupos de personas que estén trabajando en un mismo proyecto, en estos casos cuando habían proyectos grandes y se elaboran en conjunto, era incomodo estar copiando las carpetas de un programador a otro, y, en muchas ocasiones, ocurrían errores al momento de unir todas las partes, porque no se llevaba una secuencia de cada cambio, que se hacía entre los diversos programadores, también cada uno de ellos solo tenía una versión y era con la que estaban actualmente trabajando.

Los sistemas de control de versiones facilitaron todo ese mundo de problemas, guardando cada cambio para que todo programador pudiera acceder a los cambios realizados por sus compañeros, así como también poder ver las versiones anteriores por si ocurrían errores en el proyecto a futuro.

A continuación detallaremos estos Sistemas de Control de Versiones que se han convertido en herramientas indispensable en la comunidad de programadores, y que han ido evolucionado de centralizados (CVS, Subversion) a distribuidos (Git, Mercurial). Para lo cual presentamos sus definiciones y ventajas.

Esa evolución en sistemas Distribuidos, ha producido la creación de una red social de programadores llamada GitHub y GitLab. Lo cual ha beneficiado tanto a principiantes como a experimentados, que pueden compartir ideas y códigos, mejorando y complementando sus proyectos, siendo de acceso libre para todos.



OBJETIVOS

GENERAL

- Definir el concepto y clasificación de los Sistemas de Control de Versiones, brindando ejemplos, y destacando las ventajas que proporciona éstas herramientas a los programadores.

ESPECÍFICOS

- Enumerar las Características de los Sistemas de Control de versiones.
- Destacar Ventajas y Desventajas de los Sistemas de Control de versiones Centralizados como Distribuidos.
- Proporcionar Ejemplos de Sistemas de Control de Versiones tanto Centralizados como Distribuidos.
- Presentar el Software Git como alternativa para llevar el control de los proyectos, para conocerlo y aprender a implementarlo.
- Comparar las Plataformas web para distribuir y compartir proyectos usando el Software de Control de versiones GIT.

MARCO TEÓRICO

1. SISTEMAS DE CONTROL DE VERSIONES.

A. DEFINICIÓN

El control de versiones es un sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo de tal manera que sea posible recuperar versiones específicas más adelante.

Los sistemas de control de versiones han ido evolucionando a lo largo del tiempo y podemos clasificarlos en tres tipos: Sistemas de Control de Versiones Locales, Centralizados y Distribuidos. (jointDeveloper, 2017)

Se llama control de versiones a la gestión de los diversos cambios que se realizan sobre los elementos de algún producto o una configuración del mismo. Una versión, revisión o edición de un producto, es el estado en el que se encuentra el mismo en un momento dado de su desarrollo o modificación.

Aunque un sistema de control de versiones puede realizarse de forma manual, es muy aconsejable disponer de herramientas que faciliten esta gestión dando lugar a los llamados sistemas de control de versiones o VCS (del inglés Version Control System). Estos sistemas facilitan la administración de las distintas versiones de cada producto desarrollado, así como las posibles especializaciones realizadas (por ejemplo, para algún cliente específico). Ejemplos de este tipo de herramientas son entre otros: CVS, Subversion, SourceSafe, ClearCase, Darcs, Bazaar, Plastic SCM, Git, SCCS, Mercurial, Perforce, Fossil SCM, Team Foundation Server.

El control de versiones se realiza principalmente en la industria informática para controlar las distintas versiones del código fuente dando lugar a los sistemas de control de código fuente o SCM (siglas del inglés Source Code Management). Sin embargo, los mismos conceptos son aplicables a otros ámbitos como documentos, imágenes, sitios web, etc.

B. CARACTERÍSTICAS DE UN SISTEMA DE CONTROL DE VERSIONES

Un sistema de control de versiones debe proporcionar:

- Mecanismo de almacenamiento de los elementos que deba gestionar (ej. archivos de texto, imágenes, documentación...).
- Posibilidad de realizar cambios sobre los elementos almacenados (ej. modificaciones parciales, añadir, borrar, renombrar o mover elementos).
- Registro histórico de las acciones realizadas con cada elemento o conjunto de elementos (normalmente pudiendo volver o extraer un estado anterior del producto).

Aunque no es estrictamente necesario, suele ser muy útil la generación de informes con los cambios introducidos entre dos versiones, informes de estado, marcado con nombre identificativo de la versión de un conjunto de ficheros, etc. (Wikipedia C. d., 2019)

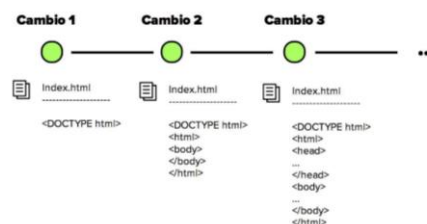


Figura 1. Línea de Tiempo de Cambios Realizados en un Archivo

C. TIPOS DE SISTEMAS DE CONTROL DE VERSIONES

i. *Sistemas de Control de Versiones Centralizados.*

En un sistema de control de versiones centralizado todos los ficheros y sus versiones están almacenados en un único directorio de un servidor. Todos los desarrolladores que quieran trabajar con esas fuentes deben pedirle al sistema de control de versiones una copia local para poder añadir o modificar ficheros. En ella realizan todos sus cambios y una vez hechos, se informa al sistema de control de versiones para que guarde las fuentes modificadas como una nueva versión. Es decir, un sistema de control de versiones centralizado funciona según el paradigma clásico cliente-servidor. (Recio Quijano, Durante Lerate, Pastrana González, & Sales Montes, s.f.)

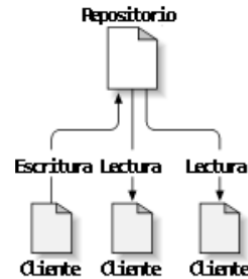


Figura 2. Sistema Cliente - Servidor Típico

Cuando se desarrollaron los sistemas de control de versiones, muchos de ellos seguían el esquema cliente-servidor, con lo que el desarrollo de las herramientas está claramente diferenciado entre dos elementos clave:

- El cliente, que es la aplicación que se conecta al servidor y mantiene una copia local del repositorio, así como se encarga, generalmente de la corrección de conflictos y la mayoría de lógica del sistema.
- El servidor, que es el sistema que toma los datos del cliente y los almacena a espera de una o múltiples peticiones de esos datos. Este sistema se encargaría del almacenamiento de las versiones, control de concurrencia, bloqueos y otras características parecidas a las de las bases de datos.

a) Ventajas De Sistemas Centralizados

- El sistema servidor es un repositorio, como los que mantienen los clientes, pero perfectamente sincronizado y sin que dé lugar a conflictos. Dicho de otro modo: es la copia maestra de los datos.
- Cuando un sistema web quiere hacer un listado, puede tomar los datos de este servidor y siempre serán fiables, con lo que no tendrá que resolver conflictos ni incongruencias.
- Una copia local debe poder mezclarse con el repositorio central cuando queramos publicar un conjunto de cambios o cuando queramos tomar la última versión publicada en concordancia con nuestra copia local.

b) Desventajas De Sistemas Centralizados

- Es lógico que en desarrollo de software aparezcan ramificaciones, versiones, etiquetas, o similares, a modo de tener varias copias de (secciones del) proyecto según nos interese. Estas ramificaciones están en el servidor y en algunos casos puede llegar a ser muy costosa su diferenciación.

ii. *Sistemas de Control de Versiones Distribuidos.*

Los Sistemas de Control de Versiones Distribuidos son herramientas de control de versiones que toman un enfoque punto a punto (peer-to-peer), al contrario de los centralizados que toman un enfoque cliente-servidor. Con ellos solo se mantiene una copia local del repositorio, pero cada equipo se convierte en un repositorio del resto de usuarios. Esto permite una fácil bifurcación en ramas y un manejo más flexible del repositorio.

En este sistema existen pocos comandos que no trabajen directamente con el propio disco duro, lo que lo convierte en una herramienta más rápida que un Sistema de Control de Versiones Centralizado. Cuando hayamos decidido que estamos listos, podemos enviar los cambios a los repositorios de los otros usuarios.

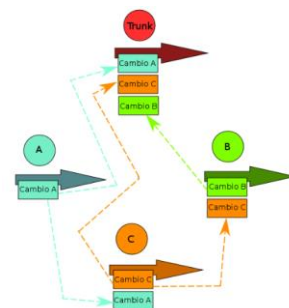


Figura 3. Funcionamiento de los Sistemas de Control de Versiones Distribuidos

Otra de las ventajas de trabajar de forma local es que puedes trabajar sin conexión a la red.

En este sistema es importante la diferencia entre ramas de desarrollo (destinadas a pruebas, inestables, etc...) y la rama principal (trunk/master), en principio la actualización del repositorio se llevará a cabo de la rama principal y no del resto de ramas, aunque últimas sí que pueden actualizarse en repositorios externos.

a) Ventajas De Sistemas Distribuidos

- Necesita menos veces estar conectado a la red para hacer operaciones. Esto produce una mayor autonomía y una mayor rapidez.
- Aunque se caiga el repositorio remoto la gente puede seguir trabajando
- Al hacer de los distintos repositorio una réplica local de la información de los repositorios remotos a los que se conectan, la información está muy replicada y por tanto el sistema tiene menos problemas en recuperarse si por ejemplo se quema la máquina que tiene el repositorio remoto. Por tanto hay menos necesidad de *backups*. Sin embargo, los *backups* siguen siendo necesarios para resolver situaciones en las que cierta información todavía no haya sido replicada.
- Permite mantener repositorios centrales más limpios en el sentido de que un usuario puede decidir que ciertos cambios realizados por él en el repositorio local, no son relevantes para el resto de usuarios y por tanto no permite que esa información sea accesible de forma pública, ya sea porque contiene versiones inestables, en proceso de codificación o con etiquetas personalizadas y de uso exclusivo del usuario.
- El servidor remoto requiere menos recursos que los que necesitaría un servidor centralizado ya que gran parte del trabajo lo realizan los repositorios locales.
- Al ser los sistemas distribuidos más recientes que los sistemas centralizados, y al tener más flexibilidad por tener un repositorio local y otro u otros remotos, estos sistemas han sido diseñados para hacer fácil el uso de ramas (creación, evolución y fusión) y poder aprovechar al máximo su potencial. Por ejemplo se pueden crear ramas en el repositorio remoto para corregir errores o crear funcionalidades nuevas. Pero también se pueden crear ramas en los repositorios locales para que los usuarios puedan hacer pruebas y dependiendo de los resultados fusionarlos con el desarrollo principal o no. Las ramas dan una gran flexibilidad en la forma de trabajo. (Wikipedia C. d., 2019)
- No existe una copia de referencia del código, solo copias de trabajo.
- Las operaciones suelen ser más rápidas al no tener que comunicarse con un servidor central.
- Cada copia de trabajo es un tipo de respaldo del código base.
- Se eliminan los problemas de latencia de la red.
- La creación y fusión de ramas es más fácil, porque cada desarrollador tiene su propia rama

b) Desventajas De Sistemas Distribuidos

1. Todavía se necesita un sistema de backup. No hay que fiarse de que el backup reside en otro usuario, ya que este puede no admitirte más, o estar inactivo mientras que yo tengo cambios hechos, por lo que todavía será necesario un servidor central donde realizar los backups.
2. Realmente no hay una última versión. Si no hay un repositorio central no hay manera de saber cuál es la última versión estable del producto.
3. Realmente no hay números de versión. Cada repositorio tiene sus propios números de revisión dependiendo de los cambios. En lugar de eso, la gente pide la última versión del GUID (número de versión) concreto, aunque cabe la posibilidad de etiquetar cada versión.
4. En los sistemas distribuidos hay menos control a la hora de trabajar en equipo ya que no se tiene una versión centralizada de todo lo que se está haciendo en el proyecto.
5. En los sistemas centralizados las versiones vienen identificadas por un número de versión. Sin embargo en los sistemas de control de versiones distribuidos no hay números de versión, ya que cada repositorio tendría sus propios números de revisión dependiendo de los cambios. En lugar de eso cada versión tiene un identificador al que se le puede asociar una etiqueta (*tag*). (Wikipedia C. d., 2019)

D. EJEMPLOS DE SISTEMAS DE VERSIONES CENTRALIZADOS

a) CVS

El Concurrent Versions System (también conocido como Concurrent Versioning System) es una aplicación informática bajo licencia GPL que implementa un sistema de control de versiones. Se encarga de mantener el registro de todo el trabajo y los cambios en los fi-cheros (código fuente principalmente) que conforman un proyecto y permite que distintos desarrolladores (potencialmente situados a gran distancia) colaboren.

Como sistema de control de versiones centralizado, CVS utiliza una arquitectura cliente-servidor como el descrito en la introducción. Típicamente, cliente y servidor se conectan utilizando Internet, pero con el sistema CVS el cliente y servidor pueden estar en la misma máquina. El sistema CVS tiene la tarea de mantener el registro de la historia de las versiones del programa de un proyecto solamente con desarrolladores locales.

Originalmente, el servidor utilizaba un sistema operativo similar a Unix, aunque en la actualidad existen versiones de CVS en otros sistemas operativos, por lo que los clientes CVS pueden funcionar en cualquiera de los sistemas operativos más difundidos. Varios clientes pueden sacar copias del proyecto al mismo tiempo. Posteriormente, cuando actualizan sus modificaciones, el servidor trata de acoplar las diferentes versiones. Si esto falla, por ejemplo debido a que dos clientes tratan de cambiar la misma línea en un archivo en particular, entonces el servidor deniega la segunda actualización e informa al cliente sobre el conflicto, que el usuario deberá resolver manualmente. Si la operación de ingreso tiene éxito, entonces los números de versión de todos los archivos implicados se incrementan automáticamente, y el servidor CVS almacena información sobre la actualización, que incluye una descripción suministrada por el usuario, la fecha y el nombre del autor y sus archivos log.

Entre otras funcionalidades, los clientes pueden también comparar diferentes versiones de archivos, solicitar una historia completa de los cambios, o sacar una "foto" histórica del proyecto tal como se encontraba en una fecha determinada o en un número de revisión determinado. Los clientes también pueden utilizar la orden de actualización con el fin de tener sus copias al día con la última versión que se encuentra en el servidor. Esto elimina la necesidad de repetir las descargas del proyecto completo.

CVS también puede mantener distintas ramas de un proyecto. Por ejemplo, una versión difundida de un proyecto de programa puede formar una rama y ser utilizada para corregir errores. Todo esto se puede llevar a cabo mientras la versión que se encuentra actualmente en desarrollo y posee cambios mayores con nuevas características se encuentre en otra línea formando otra rama separada.



b) SUBVERSIÓN

SVN es conocido así por ser el nombre del cliente, el software en sí es llamado SUBVERSIÓN.

SUBVERSIÓN es un sistema centralizado para compartir información que fue diseñado como reemplazo de CVS.

La parte principal de SUBVERSIÓN es el repositorio, el cual es un almacén central de datos. El repositorio guarda información en forma de árbol de archivos. Un número indeterminado de clientes puede conectarse al repositorio para leer (el cliente recibe información de otros) o escribir (un cliente pone a disposición de otros la información) en esos archivos



Ventajas

- Se sigue la historia de los archivos y directorios a través de copias y renombrados.
- Las modificaciones (incluyendo cambios a varios archivos) son atómicas.
- La creación de ramas y etiquetas es una operación más eficiente.
- Tiene coste de complejidad constante ($O(1)$) y no lineal ($O(n)$) como en CVS.
- Se envían sólo las diferencias en ambas direcciones (en CVS siempre se envían al servidor archivos completos). Puede ser servido mediante Apache, sobre WebDAV/DeltaV.
- Esto permite que clientes WebDAV utilicen SUBVERSIÓN en forma transparente.
- Maneja eficientemente archivos binarios (a diferencia de CVS que los trata internamente como si fueran de texto). Permite selectivamente el bloqueo de archivos. Se usa en archivos binarios que, al no poder fusionarse fácilmente, conviene que no sean editados por más de una persona a la vez.
- Cuando se usa integrado a Apache permite utilizar todas las opciones que este servidor provee a la hora de autenticar archivos (SQL, LDAP, PAM, etc.)

Carencias

- El manejo de cambio de nombres de archivos no es completo. Lo maneja como la suma de una operación de copia y una de borrado.
- No resuelve el problema de aplicar repetidamente parches entre ramas, no facilita el llevar la cuenta de qué cambios se han trasladado. Esto se resuelve siendo cuidadoso con los mensajes de commit. Esta carencia será corregida en la próxima versión (1.5)

E. EJEMPLOS DE SISTEMAS DE VERSIONES DISTRIBUIDOS

a) Mercurial

Mercurial es un sistemas de control de versiones distribuido que ofrece, entre otras cosas, "una completa ""indexación cruzada"" de ficheros y conjuntos de cambios; unos protocolos de sincronización SSH y HTTP eficientes respecto al uso de CPU y ancho de banda; una fusión arbitraria entre ramas de desarrolladores; una interfaz web autónoma integrada; [portabilidad a] UNIX, MacOS X, y Windows" y más (la anterior lista de características ha sido parafraseada del sitio web de Mercurial). (EcuRed, s.f.)



Mackall hizo pública la existencia de Mercurial el 19 de abril de 2005. El estímulo que llevó a esto fue el anuncio de Bitmover, publicado anteriormente aquel mismo mes, informando que retirarían la versión gratuita de BitKeeper. Se había estado usando BitKeeper debido a los requisitos de control de versiones del proyecto del núcleo Linux. Mackall decidió escribir un sistema de control distribuido de versiones como sustituto para usarlo con el núcleo Linux. Este proyecto comenzó aproximadamente al mismo tiempo que otro denominado git, iniciado por el propio Linus Torvalds con objetivos similares.

El proyecto Linux decidió usar Git en lugar de Mercurial. Sin embargo, muchos otros proyectos usan este último. (Wikipedia, s.f.)



b) GIT

GIT es uno de los sistemas de control de versiones distribuidos más usados.

Fue creado por Linus Torvalds y buscaba un sistema que cumpliera 4 requisitos básicos:

1. No parecido a CVS
2. Distribuido
3. Seguridad frente a corrupción, accidental o intencionada
4. Gran rendimiento en las operaciones



GIT está escrito en C y en gran parte fue construido para trabajar en el kernel de Linux, lo que quiere decir que desde el primer día ha tenido que mover de manera efectiva repositorios de gran tamaño.

Cada usuario tiene una copia completa del servidor principal, y cualquiera de ellas podría ser recuperada para reemplazarlo en caso de caída o corrupción. Básicamente, no hay un punto de fallo único con git a no ser que haya un punto único.

F. FUNDAMENTOS DE GIT

La principal diferencia entre Git y cualquier otro VCS (Subversion y compañía incluidos) es cómo Git modela sus datos. Conceptualmente, la mayoría de los demás sistemas almacenan la información como una lista de cambios en los archivos. Estos sistemas (CVS, Subversion, Perforce, Bazaar, etc.) modelan la información que almacenan como un conjunto de archivos y las modificaciones hechas sobre cada uno de ellos a lo largo del tiempo.

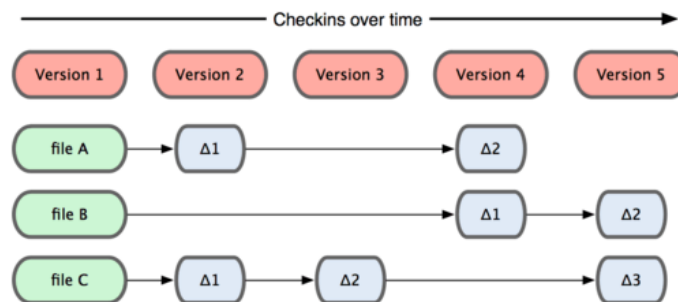


Fig. 4. Otros sistemas tienden a almacenar los datos como cambios de cada archivo respecto a una versión base.

Git no modela ni almacena sus datos de este modo. En cambio, Git modela sus datos más como un conjunto de instantáneas de un mini sistema de archivos. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea. Para ser eficiente, si los archivos no se han modificado, Git no almacena el archivo de nuevo, sólo un enlace al archivo anterior idéntico que ya tiene almacenado. Git modela sus datos más.

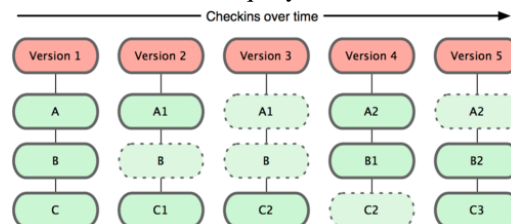


Fig. 5. Git almacena la información como instantáneas del proyecto a lo largo del tiempo.



Esta es una distinción importante entre Git y prácticamente todos los demás VCSs. Hace que Git reconsidere casi todos los aspectos del control de versiones que muchos de los demás sistemas copiaron de la generación anterior. Esto hace que Git se parezca más a un mini sistema de archivos con algunas herramientas tremendamente potentes construidas sobre él, que a un VCS.

La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para operar. Por lo general no se necesita información de ningún otro ordenador de tu red. Si estás acostumbrado a un CVCS donde la mayoría de las operaciones tienen esa sobrecarga del retardo de la red, este aspecto de Git te va a hacer pensar que los dioses de la velocidad han bendecido Git con poderes sobrenaturales. Como tienes toda la historia del proyecto ahí mismo, en tu disco local, la mayoría de las operaciones parecen prácticamente inmediatas.

Por ejemplo, para navegar por la historia del proyecto, Git no necesita salir al servidor para obtener la historia y mostrártela, simplemente la lee directamente de tu base de datos local. Esto significa que ves la historia del proyecto casi al instante. Si quieres ver los cambios introducidos en un archivo entre la versión actual y la de hace un mes, Git puede buscar el archivo hace un mes y hacer un cálculo de diferencias localmente, en lugar de tener que pedirle a un servidor remoto que lo haga, u obtener una versión antigua desde la red y hacerlo de manera local.

Esto también significa que hay muy poco que no puedas hacer si estás desconectado o sin VPN. Si te subes a un avión o a un tren y quieres trabajar un poco, puedes confirmar tus cambios felizmente hasta que consigas una conexión de red para subirlos. Si te vas a casa y no consigues que tu cliente VPN funcione correctamente, puedes seguir trabajando. En muchos otros sistemas, esto es imposible o muy doloroso. En Perforce, por ejemplo, no puedes hacer mucho cuando no estás conectado al servidor; y en Subversion y CVS, puedes editar archivos, pero no puedes confirmar los cambios a tu base de datos (porque tu base de datos no tiene conexión). Esto puede no parecer gran cosa, pero te sorprendería la diferencia que puede suponer.

Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda deshacer, o que de algún modo borre información. Como en cualquier VCS, puedes perder o estropear cambios que no has confirmado todavía; pero después de confirmar una instantánea en Git, es muy difícil de perder, especialmente si envías (push) tu base de datos a otro repositorio con regularidad.

Ahora presta atención. Esto es lo más importante a recordar acerca de Git si quieres que el resto de tu proceso de aprendizaje prosiga sin problemas. Git tiene tres estados principales en los que se pueden encontrar tus archivos: confirmado (committed), modificado (modified), y preparado (staged). Confirmado significa que los datos están almacenados de manera segura en tu base de datos local. Modificado significa que has modificado el archivo pero todavía no lo has confirmado a tu base de datos. Preparado significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación. Esto nos lleva a las tres secciones principales de un proyecto de Git: el directorio de Git (Git directory), el directorio de trabajo (working directory), y el área de preparación (staging area). (Fig 6)

El directorio de Git es donde Git almacena los metadatos y la base de datos de objetos para tu proyecto. Es la parte más importante de Git, y es lo que se copia cuando clonas un repositorio desde otro ordenador.

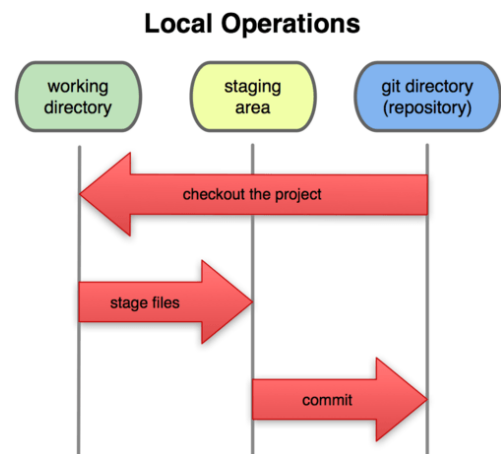


Fig. 6. Directorio de trabajo, área de preparación y directorio de Git

El directorio de trabajo es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que los puedas usar o modificar.

El área de preparación es un sencillo archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación. A veces se le denomina índice, pero se está convirtiendo en estándar el referirse a ella como el área de preparación.

El flujo de trabajo básico en Git:

1. Modificas una serie de archivos en tu directorio de trabajo.
2. Preparas los archivos, añadiéndolos a tu área de preparación.
3. Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación, y almacena esas instantáneas de manera permanente en tu directorio de Git.

Si una versión concreta de un archivo está en el directorio de Git, se considera confirmada (committed). Si ha sufrido cambios desde que se obtuvo del repositorio, pero ha sido añadida al área de preparación, está preparada (staged). Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está modificada (modified).

G. SERVICIOS WEB DE CONTROL DE VERSIONES BASADOS EN GIT

a) GitHub

Github es más que un sitio de alojamiento: se ha convertido en una red social de código, donde encontramos a otros desarrolladores y donde es muy fácil bifurcar y contribuir, creándose una comunidad en torno a GIT y los que proyectos en los que la gente lo usa.



b) GitLab



Gitlab es un servicio web de control de versiones y desarrollo de software colaborativo basado en Git. Además de gestor de repositorios, el servicio ofrece también alojamiento de wikis y un sistema de seguimiento de errores, todo ello publicado bajo una Licencia de código abierto.

Fue escrito por los programadores ucranianos Dmitry Zaporozhets y Valery Sizov en el lenguaje de programación Ruby. La compañía, GitLab Inc., cuenta con un equipo de 150 miembros y más de 1400 usuarios. Es usado por organizaciones como la NASA, el CERN, IBM o Sony.

H. DIFERENCIAS ENTRE GITHUB Y GITLAB

GitHub y GitLab son plataformas donde los usuarios pueden mantener repositorios de código. Son dos opciones similares en algunos aspectos, pero diferentes en otros. Hasta ahora uno de las principales diferencias residía en el precio. En el caso de la primera opción que mencionamos, costaba dinero crear repositorios privados. Esto ha cambiado recientemente, como vimos en un artículo anterior. Ahora GitHub permite crear repositorios privados y gratuitos. Algo que sin duda ha sido bien recibido por la comunidad. En este artículo queremos mostrar las principales diferencias que existen entre GitHub y GitLab. Entre sus diferencias



i. *Colaboradores por repositorio*

Como hemos dicho, desde hace pocas fechas la **versión gratuita de GitHub** permite crear repositorios privados y gratuitos. Esto es algo que ya ofrecía **GitLab**. Sin embargo existen diferencias entre ambos servicios. No son muy grandes, pero sí puede hacer que algunos usuarios se decanten por una u otra, según lo que necesiten.

Esta diferencia reside en el número máximo de colaboradores por repositorio. GitHub permite crear repositorios privados de manera ilimitada, pero con un máximo de 3 colaboradores. Por su parte GitLab permite también crear repositorios privados ilimitados y gratuitos, pero en esta ocasión no hay límite de colaboradores.

ii. *Tamaño máximo*

Otra característica importante y que puede hacer que un usuario elija una u otra opción es el tamaño máximo permitido. En el caso de **GitHub ofrece hasta 1 GB** por repositorio de manera gratuita. Además, cada archivo que subamos al repositorio no debe pasar de 100 MB. En caso de superar estos límites, la plataforma nos avisará.

iii. *Alternativas a GitHub gratuitas que puedes montar en tu servidor*

Por parte de **GitLab**, existen diferencias bastante positivas. Ofrece una mayor capacidad de almacenamiento. El límite por repositorio es de **10 GB**, o sea 10 veces mayor que en el caso de GitHub.

iv. *Diferencia de usuarios*

También hay que mencionar que existen **diferencias en cómo aparece cada usuario**. Es decir, en el caso de GitHub, en la versión de pago aparece “PRO” junto al nombre de usuario. En cuanto a GitLab, no hay ninguna diferencia entre los usuarios que tengan la opción gratuita y la de pago.




Es algo que no influye realmente en el uso de las plataformas, pero que sí que puede resultar interesante conocer. En el caso de GitHub, por tanto, muestran diferencias entre los usuarios de pago y los gratuitos.

v. *GitLab: Alternativa favorita para la inminente compra de GitHub por Microsoft*

En definitiva, tanto GitLab como GitHub ofrecen un plan gratuito similar. Sin embargo para algunos usuarios puede ser interesante decantarse por una opción u otra. Especialmente hay que destacar el espacio disponible que ofrece GitLab para crear un repositorio, ya que estamos hablando de 10 veces más capacidad.

Como sabemos, GitHub fue comprado por Microsoft hace unos meses. Eso provocó que muchos usuarios buscaran una alternativa a esta plataforma. Es por ello por lo que GitLab comenzó a ganar seguidores. Muchos creían que el servicio podía empeorar o comenzar a ofrecer peores prestaciones. Sin embargo, como hemos visto ahora que ha añadido la posibilidad de crear repositorios privados gratuitos, en algunos aspectos ha mejorado. (Jiménez, 2019)

I. CUADRO COMPARATIVO SISTEMAS DE CONTROL DE VERSIONES CENTRALIZADOS Y DISTRIBUIDOS

Subversion (Centralizado) 	Git (Distribuido) 	Mercurial (Distribuido) 
Repositorio autorizado central.	Todos tienen su propio repositorio.	Plataforma independiente
Todos los cambios son guardados en una única ubicación.	Los clientes pueden hacer cambios en los repositorios y estos cambios serán locales, a menos que se sincronice con alguien más.	Las diferencias son salidas que muestran los cambios entre dos versiones del mismo archivo. En solo unos segundos, también pueden retroceder en el tiempo para determinar cómo se realizaron las revisiones.
Control de Fuente/Origen en línea: El usuario debe estar en línea para enviar cambios (hacer commit) al repositorio desde la copia que él está trabajando.	Control de Fuente/Origen sin conexión: Los clientes pueden enviar cambios a repositorios como “nuevas revisiones” aun estando desconectados.	Servidor distribuido, esto significa que cada desarrollador dispone de una copia completa del repositorio, permitiéndole el manejo de las distintas versiones sin conexión
Mayor tiempo para almacenar porque todos los datos están almacenados en un repositorio centralizado.	Extremadamente rápido Se almacena una copia completa de los datos. localmente en el sistema del cliente. Mucho menos tiempo de respuesta de red.	Rápido A medida que los desarrolladores realizan revisiones y confirmaciones, la herramienta les permite producir rápidamente diferencias entre revisiones.
La ayuda de SVN es más organizada.	Hay algo de tiempo perdido ya que es difícil obtener una referencia rápida desde la búsqueda de Git.	Los usuarios de Apache Subversion pueden acceder a un conjunto de comandos con el que están familiarizados.
Más espacio de almacenamiento: Dos copias de un archivo en el directorio de trabajo de SVN. Una copia es usada para almacenar el trabajo actual/real mientras la otra, oculta en .svn/, contiene información usada para ayudar a las operaciones (estado y commit). Cuando hay muchos archivos, hay un gran impacto en el espacio de disco en SVN comparado con Git	Menos espacio de almacenamiento: Tiene una memoria eficiente porque el formato de archivo de los datos está comprimido. Git tiene un pequeño archivo de índice para almacenar información relacionada con un archivo en particular.	Almacenamiento Su rendimiento es alto en cuanto a la velocidad de ejecución de comandos y operaciones y bajo en la gestión de espacio en disco ya que siempre Mercurial está estructurado de tal forma que siempre se añaden objetos al repositorio.
Crear ramas y trabajar en la fusión de las mismas es difícil y complejo.	Es simple y fácil usar ramas y luego fusionarlas. El directorio de trabajo de un desarrollador es en sí una rama.	Fácil de aprender y usar.
Commits secuenciales: Los datos se pierden cuando se llevan a cabo commits simultáneos de dos o más copias de trabajo.	Commits no secuenciales: Gran cantidad de usuarios pueden poner datos en el mismo repositorio. No es necesario preocuparse por pérdida de datos o la fusión inmediata de otros cambios.	

Merge (Fusión de ramas):		
La facilidad para fusionar ramas también está en SVN, pero es algo incompleta.	Los usuarios tendrán control sobre la fusión de datos en repositorios sincronizados.	En este caso no son simétricos por lo que establecerá la rama más antigua como la principal y a esta unirá la más actual.
Seguimiento de Revisions:		
SVN mantiene un registro de archivos. El historial de archivos se pierde al renombrar.	Git realiza un seguimiento de los contenidos. Incluso un pequeño cambio en el contenido se identifica como un cambio separado. Git necesita una verificación global del proyecto para determinar cambios.	En Mercurial el seguimiento de los cambios se realiza en el repositorio y en un solo archivo, minimizando los costos de acceso a disco ya que no necesita buscar el archivo en cada directorio.
Checkout Parcial: Son posibles los checkouts a nivel de subdirectorio.	Solo Checkout completo: Git no permite hacer checkout a un subdirectorio. El usuario tendrá que hacer checkout a todo el repositorio.	
Usabilidad simple: Simple de aprender: create, commit y checkout. "Repositorio master" central único.	Usabilidad compleja: Dos formas de crear repositorios: -Checkout vs. Clone -Commit vs. Push Se debe saber qué comandos trabajan localmente cuáles trabajan con el servidor. Git tiene más conceptos y más comandos. Muchos comandos de Git son crípticos y los mensajes de error son poco amigables con el usuario.	La manera más sencilla pero más lenta de realizar branches es creando nuevos clones del repositorio. Otra forma es utilizar tags o bookmarks. Esta estrategia permite realizar un seguimiento del desarrollo de manera ligera y rápida. La última manera de crear branches es la más sencilla y rápida. Simplemente consiste en hacer updates y commits.
GRATIS		

J. CUANDO ELEGIR GIT Y CUANDO SUBVERSION

Debes decantarte por Git cuando...	Subversion será la opción indicada, si...
<ul style="list-style-type: none"> ...no quieres depender de una conexión de red permanente, pues quieres trabajar en tu proyecto desde cualquier lugar. ...quieres seguridad en caso de fallo o pérdida de los repositorios principales. ...no necesitas contar con permisos especiales de lectura y escritura para los diferentes directorios (aunque, de ser así, será posible y complejo implementarlo). ...la transmisión rápida de los cambios es una de tus prioridades. 	<ul style="list-style-type: none"> ...necesitas permisos de acceso basados en rutas de acceso para las diferentes áreas de tu proyecto. ...deseas agrupar todo tu trabajo en un solo lugar. ...trabajas con numerosos archivos binarios de gran tamaño. ...también quieres guardar la estructura de los directorios vacíos (estos son rechazados por Git, debido a que no contienen ningún tipo de contenido).

K. GUIA BÁSICA DE SUBVERSIÓN SVN (CDMON., 2018)

Para utilizar el cliente SVN en **cdmon**, primero debes activar el acceso por SSH en el Panel de control y tener habilitada la opción de SVN.

i. Copiar el repositorio

Para utilizar el control de versiones debes tener un repositorio SVN creado en algún servidor remoto. Para conectar y copiar el contenido del repositorio puedes hacerlo con el comando checkout.

- `svn checkout URL-repositorio`

Ejemplo:

- `svn checkout https://svn-cdmon.googlecode.com/svn/trunk/ svn-cdmon --username info@cdmon.com`

Esa orden crea el directorio que contiene los ficheros.

ii. Añadir fichero al repositorio

Cuando tengas un fichero listo para añadir al repositorio utiliza el comando `add`.

- `svn add fichero`

Ejemplo:

- `svn add test.html`

El fichero está añadido pero no se modifica el repositorio local hasta que se realice el comando `commit`.

- `svn commit -m fichero`

Ejemplo:

- `svn commit -m test.html`

Con el comando `'up'` actualizas tu copia local con la última versión del repositorio.

Ejemplo:

- `svn up test.html`

No todos los repositorios disponen de permisos para poder efectuar cambios. Tienes que asegurarte que dispones de los permisos necesarios para publicar en el servidor donde tienes el repositorio remoto.

iii. Algunos comandos de SVN

- **import:** añadir ficheros
- **checkout:** clonar un repositorio a un directorio
- **commit:** guardar los cambios al repositorio
- **diff:** mostrar las diferencias entre versiones
- **log:** mostrar el histórico de commit realizados
- **merge:** unir dos o más ramas de desarrollo
- **mv:** mover o renombrar un directorio
- **update:** recoger los ficheros de otro repositorio
- **up:** actualizar la copia local del fichero
- **rm:** eliminar ficheros del árbol de desarrollo
- **status:** mostrar el estado de la rama de trabajo actual



L. GUIA BÁSICA DE GIT. – COMANDOS. (JUAN JOSÉ LOZANO GÓMEZ, 2019)

i. Comandos Git para crear un repositorio local

Para crear un repositorio local ejecuta el siguiente comando:

```
git init
```

Una vez creado, debes configurar el nombre y la dirección de correo electrónico con los que «firmarás» los commits del repositorio recién creado. Para ello ejecuta los siguientes comandos Git:

```
git config user.name "Tu nombre"  
git config user.email tu_direccion_de_email
```

Si quieres que esta información sirva para cualquier repositorio que configures en tu sistema, debes añadir la opción `--global`. Yo no te la recomiendo, ya que quizá trabajes en distintos proyectos/equipos y tu dirección de email puede que sea distinta. En cualquier caso, el comando Git es:

```
git config --global user.name "Tu nombre"  
git config --global user.email tu_direccion_de_email
```

ii. Integrando un repositorio remoto

Para integrar un repositorio local con un repositorio remoto debes ejecutar el siguiente comando Git:

```
git remote add origin <URL o PATH del repositorio remoto>  
#Ejemplo:  
git remote add origin https://github.com/j2logo/tutorial-flask.git
```

La palabra `origin` no es obligatoria. Es el nombre con el que identificar al repositorio remoto (puedes añadir tantos repositorios remotos como quieras).

También puedes hacer un checkout o clonar un repositorio remoto directamente. Esto te ahorrará el paso de crear el repositorio local, ya que se crea automáticamente y quedará conectado al repositorio remoto. Para ello ejecuta:

```
git clone <URL o PATH del repositorio remoto>  
#Ejemplo:  
git clone https://github.com/j2logo/tutorial-flask.git
```

iii. Comandos Git para confirmar los cambios: *add* y *commit*

Una vez que quieras confirmar los cambios en tu repositorio local hay dos pasos que debes seguir. En primer lugar debes indicar qué ficheros contienen los cambios que quieres confirmar, para lo que se usa el comando `add`:

```
git add <nombre_fichero>  
# Para incluir todos los ficheros con cambios, ejecuta:  
git add .
```

Una vez que has indicado los cambios a confirmar, es hora de hacer un `commit`:



```
git commit -m "Mensaje del commit"
```

iv. *Integrando los cambios en tu repositorio remoto*

Cuando tu cambios en el repositorio local estén listos para ser integrados en el repositorio remoto, el comando a ejecutar es:

```
git push origin <nombre_rama_local>
```

Recuerda que `origin` es el nombre que le dimos al repositorio remoto.

v. *Comandos Git para crear ramas*

Las ramas nos permiten trabajar en distintas funcionalidades a la vez y en distintas versiones de nuestra aplicación sin que los cambios afecten de unas a otras.

Para crear una rama y situarte en ella ejecuta:

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

Si quieres cambiarte a otra rama existente:

```
git checkout <nombre_otra_rama>
```

Recuerda borrar una rama cuando ya no tengas que trabajar más en ella:

```
git branch -d <nombre_rama>
```

vi. *Fusionando el contenido de una rama en otra*

Una vez que el trabajo en una rama secundaria de tu repositorio local ha terminado, debes integrar los cambios en una rama principal. Por ejemplo, de una rama `feature` a la rama `dev`. Para ello, debes hacer un merge de la rama `feature` desde `dev` de la siguiente manera:

```
# Sitúate en dev
git checkout dev
# Ahora fusiona la rama feature-1
git merge feature-1
```

El comando para integrar los cambios del repositorio remoto a tu repositorio local es:

```
git pull
```

vii. *Comandos Git para deshacer los cambios: revert*

Es posible que en alguna ocasión los cambios realizados no sirvan y quieras volver a la versión anterior de un fichero. En ese caso, ejecuta:

```
git checkout -- <nombre_fichero>
```

Para deshacer los cambios locales y commits y traer la última versión estable del repositorio remoto, los pasos son los siguientes:

```
git fetch origin
git reset --hard <nombre_rama_remota>
# Ejemplo:
git reset --hard origin/dev
```



M. EJEMPLO DE PROYECTO EN GIT USANDO GITHUB. (PRODUCTIVIDAD, 2019)

16 commits	2 branches	0 releases	2 contributors
Branch: master	New pull request	Find File	Clone or download
Berny96 Ultimo	Latest commit 54bc540 on 16 Jul		
AppTrans	Ultimo	2 months ago	
Proyecto_G14	Ultimo	2 months ago	
db.sqlite3	Creación de registros captura la fecha y hora	2 months ago	
manage.py	Creación de registros captura la fecha y hora	2 months ago	

Branch: master

CD17008/hdpproyecto

<https://github.com/cd17008/hdpproyecto.git>

Commits on Jul 16, 2019

Ultimo	54bc540	<>
Berny96 committed on 16 Jul		
Corrección de errores en labels	575385d	<>
Berny96 committed on 16 Jul		

Commits on Jul 15, 2019

Merge branch 'master' of https://github.com/cd17008/hdpproyecto	3a5188f	<>
Berny96 committed on 15 Jul		
Pequeños cambios	a7d55a8	<>
Berny96 committed on 15 Jul		
crear logout	52beaa7	<>
Reynaldo committed on 15 Jul		
crear logout	55812c4	<>
Reynaldo committed on 15 Jul		

Commits on Jul 14, 2019

URL crearResponsable	Verified 3a652a5	<>
Berny96 committed on 14 Jul		
Ver Detalle, Cambios al Home, Removido atributo dirección	b01e13b	<>
Berny96 committed on 14 Jul		
Ver Detalle, Cambios al Home	16ebae6	<>
Berny96 committed on 14 Jul		
creacion de responsable y mas cambios	9a41cd8	<>
Reynaldo committed on 13 Jul		

Commits on Jul 13, 2019

Crear Unidad, Crear Motorista, Registro Usuarios	5707b20	<>
Berny96 committed on 13 Jul		
login y creacion de puntos de control	3a7c9fc	<>
Reynaldo committed on 12 Jul		

Commits on Jul 12, 2019

con home	dab2b85	<>
unknown committed on 12 Jul		

Commits on Jul 4, 2019

Creación de registros captura la fecha y hora	4de4187	<>
Berny96 committed on 4 Jul		

Commits on Jul 1, 2019

Actualización	454dbe3	<>
cd17008 committed on 30 Jun		

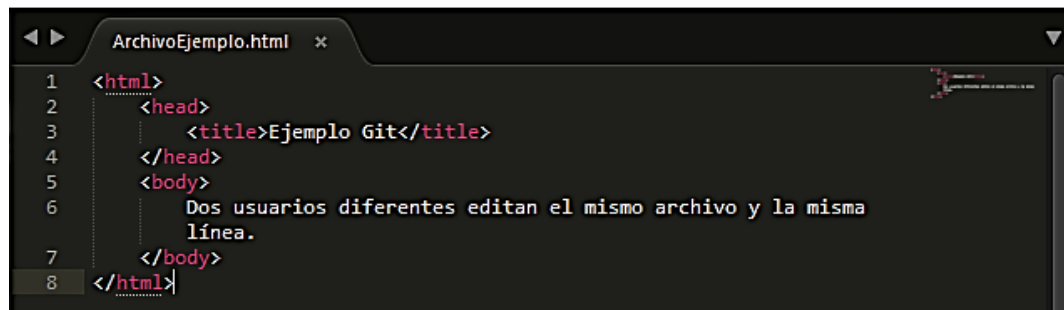
Commits on Jun 23, 2019

versión inicial del proyecto	b36e8a6	<>
cd17008 committed on 23 Jun		

N. EJEMPLO PASO A PASO DE SOLUCIÓN CONFLICTOS EN GIT USANDO GITHUB.

¿Qué ocurre cuando dos usuarios modifican (cada uno en su repositorio local) un mismo archivo y precisamente la misma línea de código, e intentan enviar los cambios al repositorio remoto?

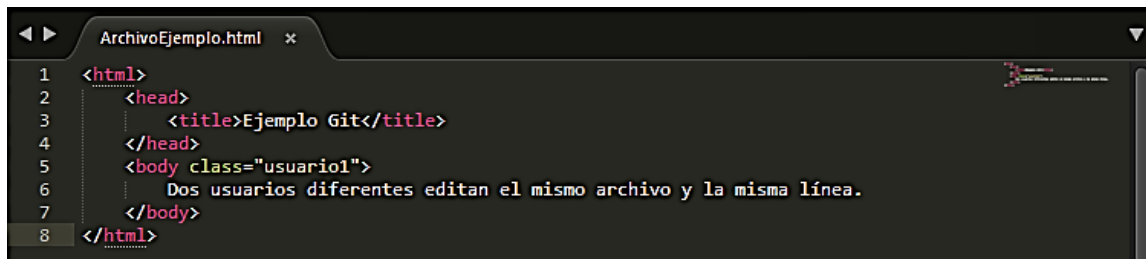
En un repositorio remoto llamado TOO115, en una rama de llamada Bernardo se tiene un archivo con nombre ArchivoEjemplo.html, cuyo contenido se muestra en la siguiente imagen:



```
1 <html>
2   <head>
3     <title>Ejemplo Git</title>
4   </head>
5   <body>
6     Dos usuarios diferentes editan el mismo archivo y la misma
7     línea.
8   </body>
9 </html>
```

Existe un usuario que está contribuyendo en TOO115, a quien se le llamará Usuario1, que ha clonado la rama Bernardo. Con esto consigue una copia de ésta en un repositorio local (es decir, la rama Bernardo y todo su contenido ha sido agregada en su propia máquina). Decide trabajar en ArchivoEjemplo.html para agregar cambios que él desea hacer. Dicho usuario modifica la línea 5 del archivo, y le agrega un atributo *class* a la etiqueta *body*, asignándole el valor de “usuario1”.

El resultado del cambio es éste:



```
1 <html>
2   <head>
3     <title>Ejemplo Git</title>
4   </head>
5   <body class="usuario1">
6     Dos usuarios diferentes editan el mismo archivo y la misma línea.
7   </body>
8 </html>
```

Ahora Usuario1 decide publicar (o subir) el cambio realizado en ArchivoEjemplo, al repositorio remoto, ya que en este momento el cambio está solamente en su repositorio local y solo es visible para él y no para otros usuarios que contribuyan en el repositorio TOO115. Para enviar sus cambios ejecuta la siguiente secuencia de comandos:

`git add .`

Este comando es el primero que debe ejecutarse para hacer un seguimiento de los archivos cuyos cambios quieren enviarse al repositorio remoto (hacer commit). El carácter “.” después de “add” identifica todos los archivos a los que se han hecho cambios y los pone en una lista especial del SCV llamada *Cambios a confirmar*. Este comando también puede ejecutarse para archivos únicos, así: `git add [nombre del archivo]`.

`git commit -m “Mensaje del commit”`

Este comando permite confirmar los cambios pendientes de confirmar, debiendo agregarse un comentario para el commit. Los archivos confirmados están listos para actualizarse hacia el repositorio remoto.

git push origin [rama]

Al ejecutar este comando los cambios confirmados se envían directamente al repositorio remoto. Debe indicarse la rama a la que se envían los cambios.

```
IsraelErazo@DESKTOP-HOATVGT MINGW32 ~/T00115 (Bernardo)
$ git add .

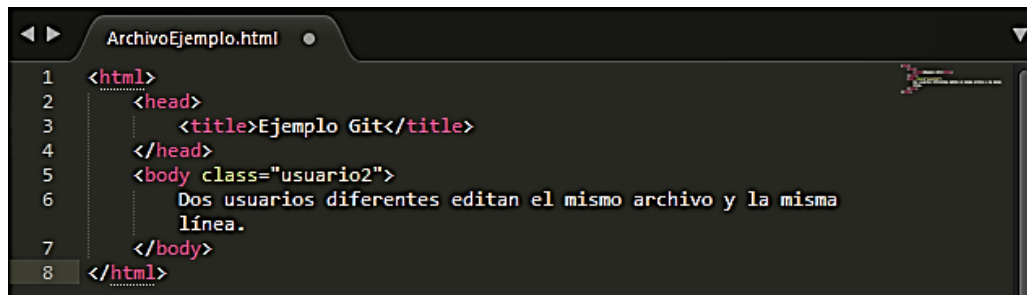
IsraelErazo@DESKTOP-HOATVGT MINGW32 ~/T00115 (Bernardo)
$ git commit -m "Clase agregada a etiqueta body"
[Bernardo 35c11c0] Clase agregada a etiqueta body
1 file changed, 1 insertion(+), 1 deletion(-)

IsraelErazo@DESKTOP-HOATVGT MINGW32 ~/T00115 (Bernardo)
$ git push origin Bernardo
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes | 35.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/gso2009/T00115.git
   38c36d3..35c11c0  Bernardo -> Bernardo

IsraelErazo@DESKTOP-HOATVGT MINGW32 ~/T00115 (Bernardo)
$ |
```

Con esto, el Usuario1 ha enviado sus cambios al repositorio remoto con éxito.

Existe otra persona agregado a TOO115 para contribuir en el desarrollo del proyecto. Este usuario se llamará Usuario2. Al igual que el Usuario1, Usuario2 ha clonado la rama Bernardo, con lo que tiene una copia local de la rama en su máquina. Usuario2 decide hacer cambios en ArchivoEjemplo.html. Después de modificarlo, el archivo queda así:



```
1 <html>
2   <head>
3     <title>Ejemplo Git</title>
4   </head>
5   <body class="usuario2">
6     Dos usuarios diferentes editan el mismo archivo y la misma
7     línea.
8   </body>
</html>
```

Posteriormente decide enviar sus cambios a la rama Bernardo del repositorio remoto TOO115, para lo cual, ejecuta la siguiente secuencia de comandos:

```
IsraelErazo@DESKTOP-HOATVGT MINGW32 ~/Documents/T00115 (Bernardo)
$

IsraelErazo@DESKTOP-HOATVGT MINGW32 ~/Documents/T00115 (Bernardo)
$ git add .

IsraelErazo@DESKTOP-HOATVGT MINGW32 ~/Documents/T00115 (Bernardo)
$ git commit -m "Atributo class de la etiqueta body"
[Bernardo cd61ecd] Atributo class de la etiqueta body
1 file changed, 1 insertion(+), 1 deletion(-)

IsraelErazo@DESKTOP-HOATVGT MINGW32 ~/Documents/T00115 (Bernardo)
$ git push origin Bernardo
To https://github.com/gso2009/T00115/
 ! [rejected]        Bernardo -> Bernardo (fetch first)
error: failed to push some refs to 'https://github.com/gso2009/T00115/'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.

IsraelErazo@DESKTOP-HOATVGT MINGW32 ~/Documents/T00115 (Bernardo)
$
```

El SCV (Sistema de Control de Versiones) le informa de un problema: Existen cambios en el repositorio remoto TOO115 que no están actualizados en el repositorio local del Usuario2. Esto se debe a los cambios hechos por Usuario1. Sin embargo el SVN le indica la solución: Ejecutar el comando git pull, con lo que se identificarán los cambios que otros usuarios han realizado en el repositorio remoto y su repositorio local será actualizado con los estos nuevos cambios. Al ejecutar git pull, Usuario2 obtiene el siguiente resultado:

```
IsraelErazo@DESKTOP-HOATVGT MINGW32 ~/Documents/TOO115 (Bernardo)
$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 2), reused 3 (delta 2), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/gso2009/TOO115
 38c36d3..35c11c0  Bernardo  -> origin/Bernardo
Auto-merging ArchivoEjemplo.html
CONFLICT (content): Merge conflict in ArchivoEjemplo.html
Automatic merge failed; fix conflicts and then commit the result.

IsraelErazo@DESKTOP-HOATVGT MINGW32 ~/Documents/TOO115 (Bernardo|MERGING)
$
```

Luego de ejecutar el comando git pull, el SCV identifica un conflicto al fusionar el archivo ArchivoEjemplo.html del repositorio remoto con ArchivoEjemplo.html del repositorio local de Usuario2: El merge (la acción de fusionar los archivos) automático falló; y solicita se solucionen los conflictos y luego hacer commit (enviar cambios) otra vez.

Esto significa que tanto Usuario1 como Usuario2 hicieron cambios en la misma línea de ArchivoEjemplo.html y, evidentemente, su contenido difiere. Cuando Usuario2 abre ArchivoEjemplo.html, se encuentra con un cambio en su contenido, líneas de código agregadas por el SCV.

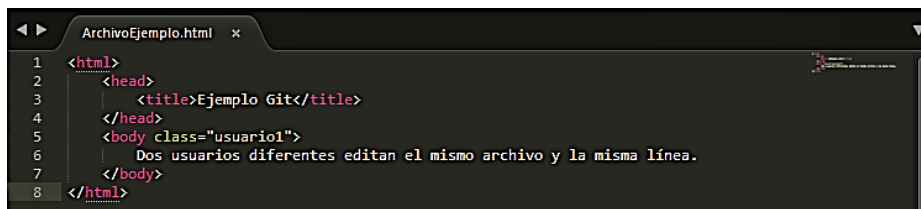


```
1 <html>
2 <head>
3 <title>Ejemplo Git</title>
4 </head>
5 <<<<<< HEAD
6 <body class="usuario2">
7 =====
8 <body class="usuario1">
9 >>>>>> 35c11c0eb78823ea6adf16ef16386d8cbac6d431
10 Dos usuarios diferentes editan el mismo archivo y la misma
11 línea.
12 </body>
13 </html>
```

Estas líneas son la 5, la 7 y la 9. El SCV agrega esas líneas para indicar el conflicto encontrado. Las líneas 5 y 9 delimitan el problema, mientras que la línea 7 separa la línea de código 6 (la línea 5 original modificada por Usuario2) de la línea 8 (la línea 5 original modificada por el Usuario1).

En este punto ambos programadores deben ponerse de acuerdo respecto a cuál de las líneas será la definitiva para el repositorio remoto.

Luego de hablar, Usuario1 y Usuario2 acordaron que la línea de código correcta era la modificada por Usuario1. Entonces Usuario2, quien fue el que tuvo el conflicto, modifica el archivo ArchivoEjemplo.html, quedando el contenido del archivo así:



```
1 <html>
2 <head>
3 <title>Ejemplo Git</title>
4 </head>
5 <body class="usuario1">
6 Dos usuarios diferentes editan el mismo archivo y la misma
7 línea.
8 </body>
9 </html>
```


Luego Usuario2 realiza de nuevo el commit, ejecutando la misma secuencia de comandos anterior, como se muestra en la imagen:

```
IsraelErazo@DESKTOP-HOATVGT MINGW32 ~/Documents/TOO115 (Bernardo|MERGING)
$ git add .

IsraelErazo@DESKTOP-HOATVGT MINGW32 ~/Documents/TOO115 (Bernardo|MERGING)
$ git commit -m "Resolviendo el conflicto"
[Bernardo c4539b0] Resolviendo el conflicto

IsraelErazo@DESKTOP-HOATVGT MINGW32 ~/Documents/TOO115 (Bernardo)
$ git push origin Bernardo
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 4 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 658 bytes | 65.00 KiB/s, done.
Total 6 (delta 4), reused 0 (delta 0)
remote: Resolving deltas: 100% (4/4), completed with 2 local objects.
To https://github.com/gs02009/TOO115/
35c11c0..c4539b0 Bernardo -> Bernardo

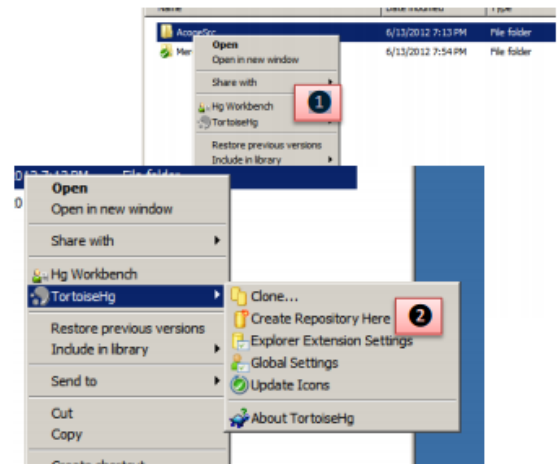
IsraelErazo@DESKTOP-HOATVGT MINGW32 ~/Documents/TOO115 (Bernardo)
$ |
```

Como puede verse, luego de hacer push (enviar los cambios al repositorio remoto), el SCV no identifica ningún problema, con lo que se deduce que el conflicto ha sido resuelto exitosamente y los cambios son enviados a la rama Bernardo en el repositorio TOO115.

O. GUIA BÁSICA DE MERCURIAL. (QUEVEDO, 2012)

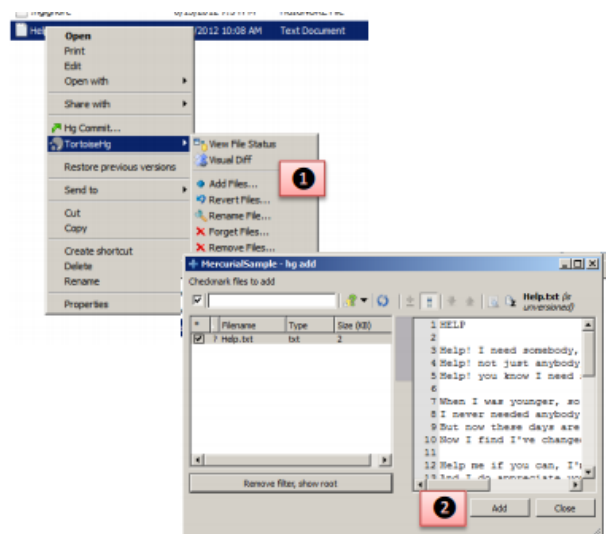
i. Creación de repositorio

1. En el directorio que desea crear un repositorio usar el botón derecho y en el menú desplegable seleccionar Tortoise HG.
 2. Seleccionar la opción de Create Repository Here.
- Se crearán un sistema de directorios y archivos que Mercurial usa para registrar los cambios y generar las copias por cada versión confirmada (committed).



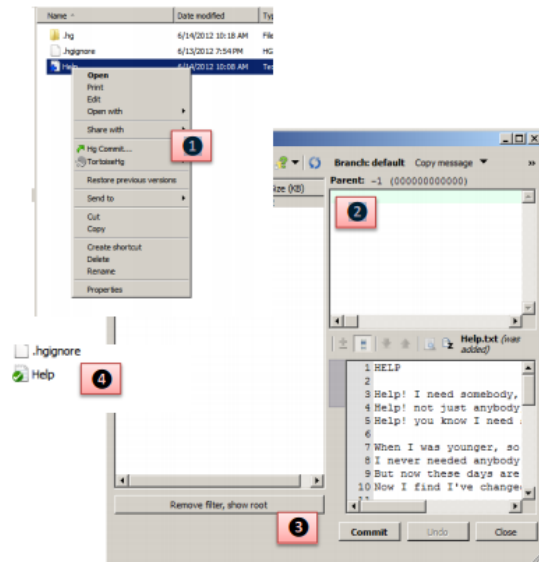
ii. Nuevos archivos

1. Se pueden crear nuevos archivos copiando o creando con cualquier editor. Al encontrar un nuevo archivo, Mercurial no hace nada hasta que el archivo se añade al control de versiones. Para añadir un archivo al control de versiones use el menú desplegado con el botón derecho y elija TortoiseHg > Add Files.
 2. Confirme con el botón Add en la pantalla que muestra el o los archivos y el contenido.
- El archivo se mostrará con el signo + de añadido. Todavía no tiene una versión en el registro de cambios porque no se hizo la confirmación (Commit).



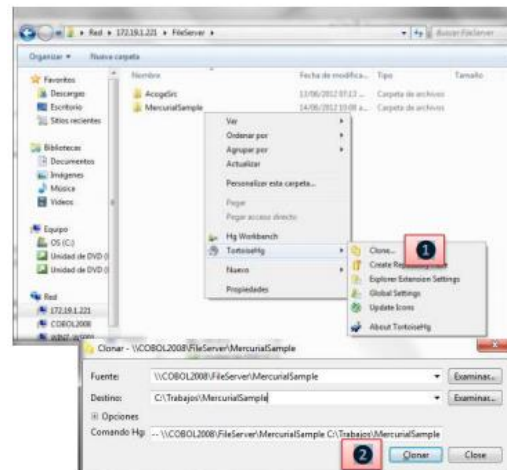
iii. Confirmación de cambios en un archivo

1. La confirmación se puede hacer sobre un archivo o sobre el repositorio (directorio). En el menú emergente de botón derecho elija Hg Commit...
2. Esta acción creará un registro histórico y se congelará una copia en el repositorio que siendo la última versión se conoce como Tip. En el campo de mensaje llenar un comentario describiendo el cambio.
3. Aceptar la confirmación con el Botón Commit en la pantalla Mercurial.
4. El ícono del archivo cambiará al símbolo ✓ que muestra a los archivos que no han tenido modificaciones desde la última confirmación.



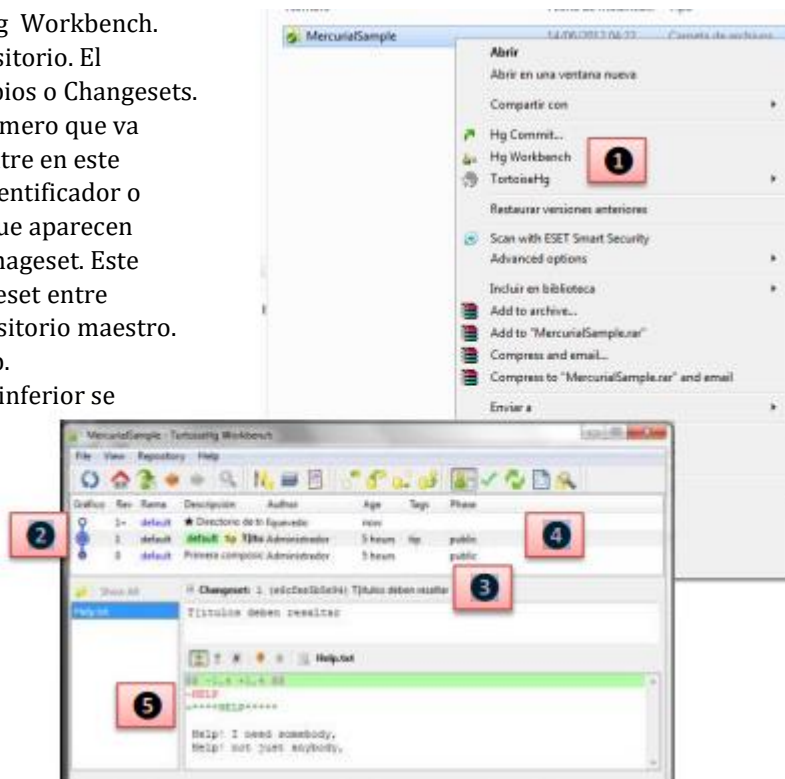
iv. Creación de copia personal de repositorio

- Desde cualquier computadora que tenga acceso al repositorio y se haya instalado los programas de mercurial se puede crear una copia controlada que permite trabajar a varias personas independientemente sin hacer modificaciones al repositorio central hasta que sean confirmadas.
 - El control de versiones de Mercurial permite saber qué modificaciones se han hecho por cada usuario y permite la sincronización de las versiones distribuidas.
1. Usando el Explorer, en la PC cliente busque el directorio de repositorio Mercurial del que desea generar una copia controlada. Elija la opción Tortoise > Clone del menú emergente del botón derecho.
 2. Se presenta la ventana de Mercurial que permite elegir el repositorio origen y el directorio que se desea convertir en su copia controlada.. Elija Clonar. Esta acción generará una copia de todos los archivos actuales en su última versión y el historial de los cambios. No es recomendable hacer una copia por los métodos del S.O. o el explorer.
- Sobre estas copias clonadas los usuarios podrán hacer sus modificaciones y crear nuevos archivos controlados. Luego revisaremos la forma de sincronizar la copia principal y las derivadas.



v. *Revisión del historial*

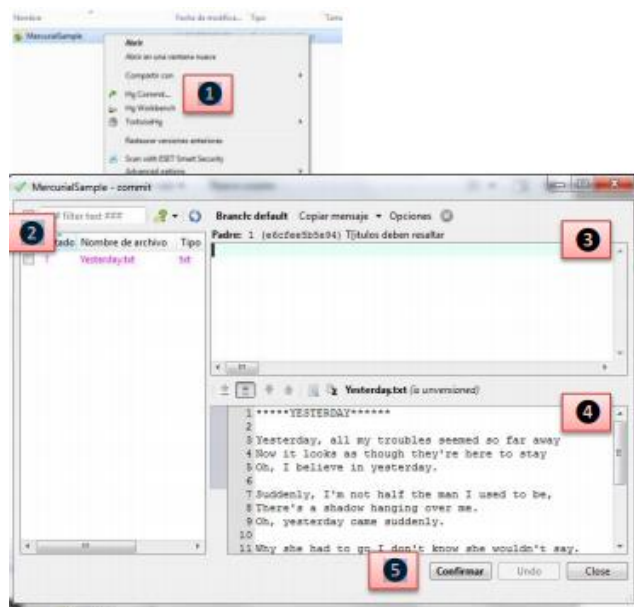
1. Desde el menú emergente del botón derecho del repositorio seleccionado, elija la opción Hg Workbench.
2. Aparece la pantalla de historial de repositorio. El historial se compone de una serie de cambios o Changesets.
3. Cada Changeset tiene un secuencia o número que va del 0 al número de cambios que se encuentre en este repositorio. El Changeset se asocia a un identificador o toquen que son 12 dígitos exadecimales que aparecen entre paréntesis después de número de Chageset. Este toquen permite identificar el mismo Chageset entre varias copias distribuidas del mismo repositorio maestro.
4. El último Chageset es conocido como tip.
5. Al seleccionar un Changeset en la parte inferior se detallan los cambios de ese Commit.



vi. *Actualizaciones locales*

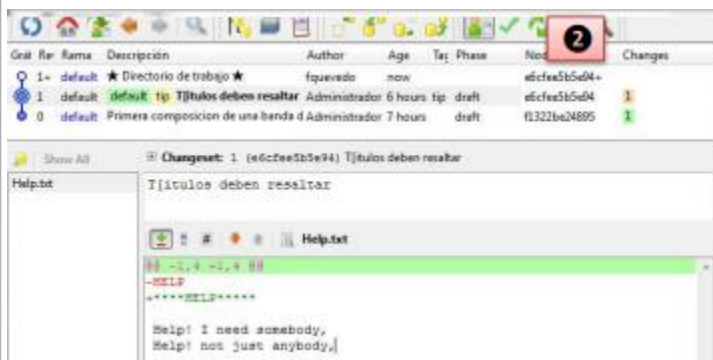
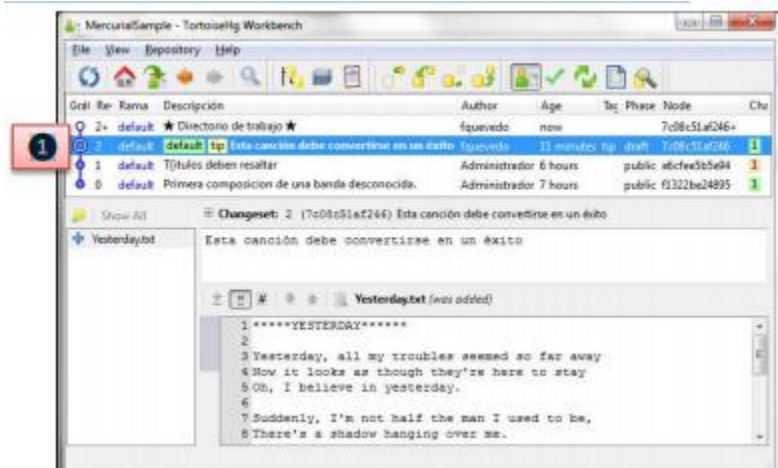
- Cuando se trabaja en un repositorio copia de un maestro, se modifican los archivos que están en el directorio de trabajo, es decir los que aparecen en los directorios de sistema de archivos normal. El historial Mercurial se guarda en los directorios .hg.
- Sólo la última modificación en el directorio de trabajo se muestran. Es decir, acumula los cambios desde el último Commit (no se genera historial con los comandos save).

1. Para confirmar los cambios seleccione HG Commit... en el menú de botón derecho.
2. Emerge el formulario de Commit del Mercurial. A la izquierda aparecen los archivos describiendo es estado en comparación al tip. Los archivos que no se añadido pueden seleccionarse (Check) para incluirlos en el Changeset.
3. En la parte superior derecha debe ingresarse la descripción de los cambios que se asociará al nuevo Chageset.
4. En la parte derecha inferior se ven las modificaciones al archivo seleccionado.
5. Seleccione el botón confirmar para guardar el nuevo Chageset. Esta acción generará el historial en el repositorio local. Tener en cuenta que el maestro no ha cambiado con este procedimiento.



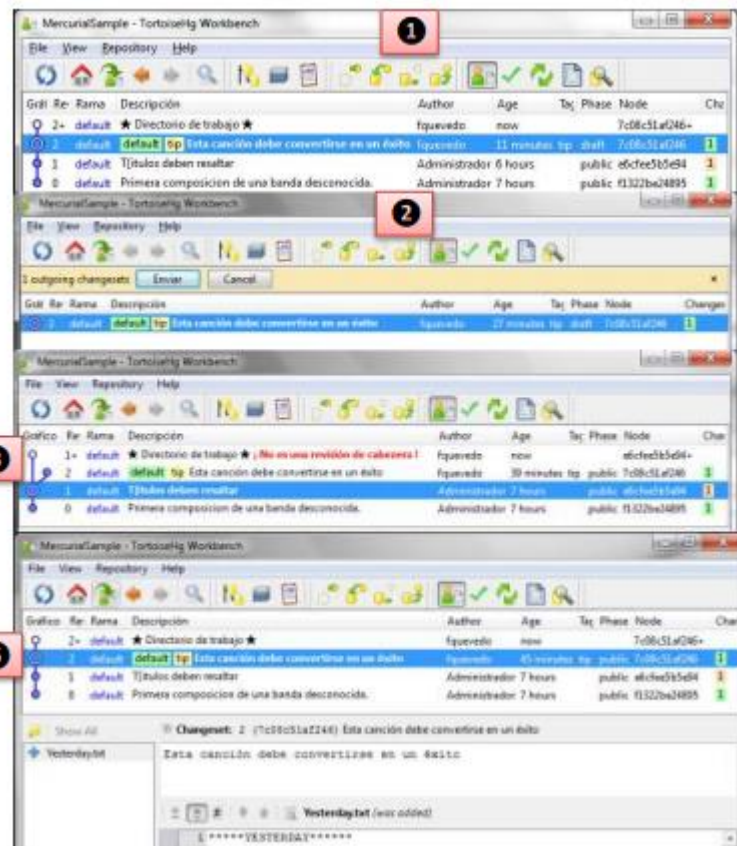
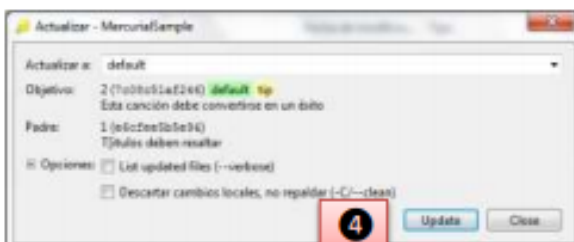
vii. Comparar repositorio local y maestro

1. El local (arriba) tiene un Chageset adicional que no aparece en el maestro (abajo).
2. Los tokens en la columna de nodo coinciden porque son los mismos Chageset .



viii. Actualizar el repositorio maestro

1. Para actualizar los cambios en el repositorio local, identifique los cambios con el botón de Detect outgoing changes....
2. Cuando se han encontrado los cambios haga el envío usando el botón de Push o Enviar.
3. Esta acción actualizará el repositorio maestro, sin embargo el directorio de trabajo no se actualizará. Y se creará una cabecera diferente donde está el nuevo tip importado.
4. Para actualizar el directorio de trabajo, en el menú de botón derecho del directorio elija la opción Update.
5. Sólo después de esta acción se alinean las versiones Y El directorio de trabajo se actualiza con los nuevos cambios.



ix. Actualizar el repositorio local

1. Para asegurarnos que estamos trabajando sobre las última versión del repositorio maestro , entramos a la pantalla de Workbench y seleccione el botón de Check incoming changes...
2. Cuando se han encontrado los cambio jale las actualizaciones usando el botón de Pull incomin changes.. o Aceptar.
3. Esta acción actualizará el repositorio local, sin embargo el directorio de trabajo no se actualizará. Y se creará una cabecera diferente donde está el nuevo tip importado.
4. Para actualizar el directorio de trabajo, en la línea de Changeset del tip elija la opción Actualizar... del menú de botón derecho.
5. Sólo después de esta acción se alinean las versiones y el directorio de trabajo se actualiza con los nuevos cambios.



CONCLUSIONES

Los Sistemas de Control de Versiones, son una herramienta indispensable para los grupos de programadores que trabajan en proyectos comunes, han evolucionado de Centralizados a Distribuidos, adquiriendo ventajas y disminuyendo el trabajo de combinar todos los sub códigos proporcionados por todos los integrantes.

La irrupción del internet, las redes sociales y ahora redes de programadores, ha posibilitado el trabajo conjunto de personas en polos opuestos del planeta, combinando conocimientos, experiencias y dando como resultado mejores proyectos, más completos y eficientes.

El Software Git, proporciona de una forma fácil, intuitiva y muy efectiva la integración de las versiones y el control de cambios, el cual es una de sus virtudes al hacer uso de ramas y combinaciones de código que permite seguir ocupando la aplicación ininterrumpida mientras se modifica.

Tanto GitHub como GitLab, proporcionan un sitio de reunión, para el control, creación y distribución de códigos de aplicaciones, donde todos pueden apoyar o aprender de los participantes.

BIBLIOGRAFÍA

- Andre, L. (s.f.). *financesonline.com*. Obtenido de <https://financesonline.com/version-control-systems/>
- Cdmon. (17 de Octubre de 2018). *ticket.cdmon.com*. Obtenido de <https://ticket.cdmon.com/es/support/solutions/articles/7000006247-manual-b%C3%A1sico-de-svn>
- EcuRed. (s.f.). https://www.ecured.cu/EcuRed:Enciclopedia_cubana. Obtenido de https://www.ecured.cu/Sistemas_de_control_de_versiones#Mercurial
- Ionos. (30 de 05 de 2016). *ionos.es/digitalguide/*. Obtenido de <https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/git-vs-svn-una-comparativa-del-control-de-versiones/>
- Jiménez, J. (10 de 01 de 2019). <https://www.redeszone.net>. Obtenido de <https://www.redeszone.net/2019/01/10/github-vs-gitlab-diferencias/>
- jointDeveloper. (28 de Enero de 2017). <https://medium.com/>. Obtenido de <https://medium.com/@jointdeveloper/sistemas-de-control-de-versiones-qu%C3%A9-son-y-por-qu%C3%A9-amarlos-24b6957e716e>
- Juan José Lozano Gómez, I. I. (14 de 02 de 2019). <https://j2logo.com>. Obtenido de <https://j2logo.com/comandos-git-principales/>
- Productividad, H. 1. (2019). <https://github.com/cd17008/hdpproyecto.git>.
- Quevedo, F. (22 de 06 de 2012). *Guia básica de Mercurial para Windows*. Obtenido de <https://es.scribd.com/document/97940742/Guia-basica-de-Mercurial-para-Windows-espanol#download>
- Recio Quijano, P., Durante Lerate, R., Pastrana González, L., & Sales Montes, N. (s.f.). *Sistema de Control de Versiones*. Obtenido de <https://rodin.uca.es/xmlui/bitstream/handle/10498/9785/trabajoSCV.pdf>
- Wikipedia. (s.f.). <https://es.wikipedia.org>. Obtenido de <https://es.wikipedia.org/wiki/Mercurial>
- Wikipedia, C. d. (29 de Julio de 2019). <https://es.wikipedia.org>. Obtenido de https://es.wikipedia.org/wiki/Control_de_versiones