



Sesión 2

Sistemas Multimedia

Git

- Con diferencia, el sistema de control de versiones **más utilizado del mundo**.
- Fue desarrollado por Linus Torvalds.
 - “I really never wanted to do source control management at all and felt that it was just about the least interesting thing in the computing world (with the possible exception of databases ;^)”.

Fuente: <https://www.linuxfoundation.org/blog/10-years-of-git-an-interview-with-git-creator-linus-torvalds/>

- **No confundir Git con Github.**
- Github es una plataforma que usa Git. Otra conocida es por ejemplo Bitbucket.

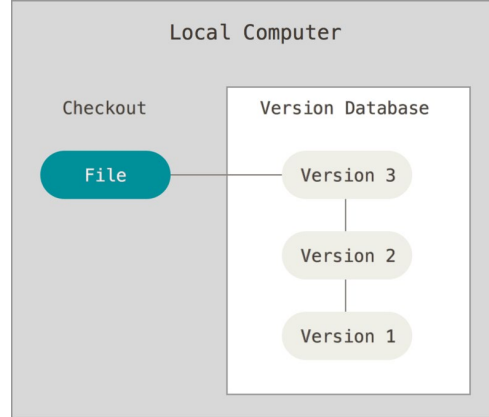
Control de versiones

- El control de versiones es un sistema que registra los cambios de archivo o archivos a lo largo del tiempo.
 - Gracias a esto, podemos recuperar versiones anteriores en un tiempo futuro.
 - Revertir los cambios cuando estos no funcionen.
 - Cuando me cargue algo y no sepa el qué pero sí cuando funcionaba puedo volver al punto en el que las cosas funcionaban.
- Todo esto sin apenas esfuerzo. Ahora bien, **hay que ser ordenado.**

Tipos de Sistemas de Control de Versiones (VCS)

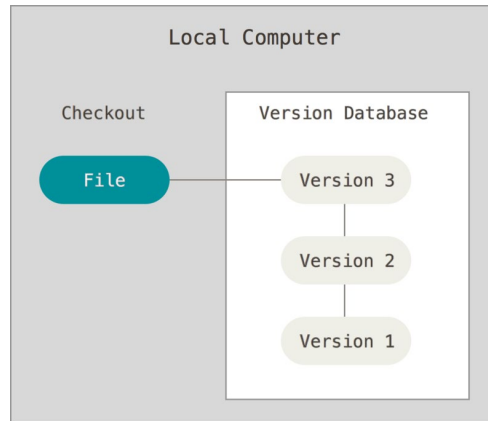
Podemos clasificar los VCS en tres tipos:

1. **Local.** Los cambios se producen y mantienen en el ordenador del desarrollador. Ejemplo: <https://www.gnu.org/software/rcs/>



Tipos de Sistemas de Control de Versiones (VCS)

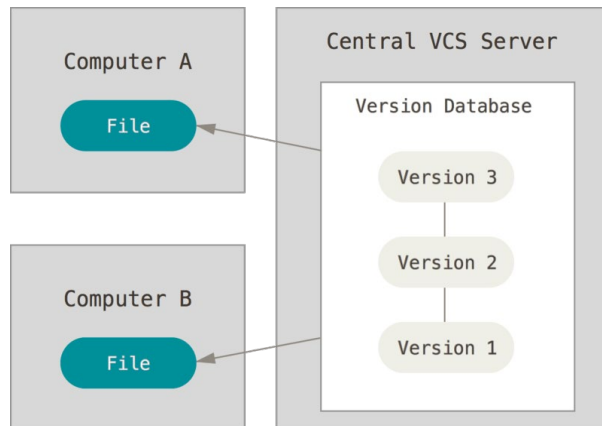
¿Y si colaboramos con más gente?



Tipos de Sistemas de Control de Versiones (VCS)

Podemos clasificar los VCS en tres tipos:

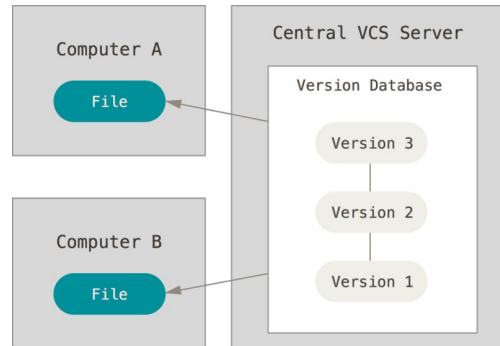
- 2. Centralizado.** Existe un servidor que contiene los archivos versionados y los clientes extraen archivos de ese lugar central. Si alguien modifica haciendo push, el resto tendrá que adaptar el código. Ejemplo: CVS, Subversion...



Tipos de Sistemas de Control de Versiones (VCS)

Ventajas:

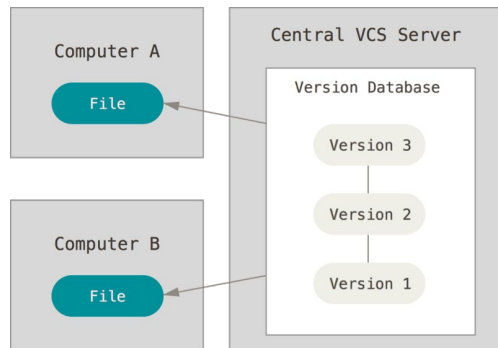
- Todos saben hasta cierto punto lo que están haciendo los demás.
- Los administradores tienen un control detallado sobre quién puede hacer qué.
- Más fácil administrar un VCS que tratar con bases de datos locales en cada cliente.



Tipos de Sistemas de Control de Versiones (VCS)

Inconvenientes:

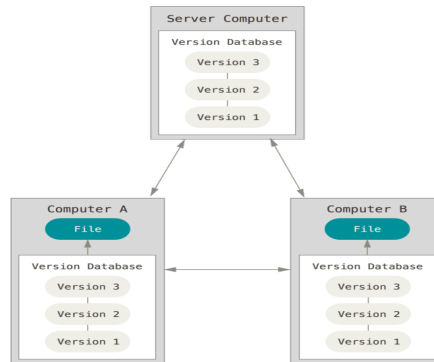
- Si ese servidor cae, durante su caída, no se puede colaborar.
- Si el disco del servidor central se estropea (y no tenemos copia de seguridad) perderemos todo.



Tipos de Sistemas de Control de Versiones (VCS)

Podemos clasificar los VCS en tres tipos:

- 3. Distribuido.** Los clientes tienen una instantánea del repositorio completo, incluido su historial. Si el servidor cae, un cliente de los que estaba colaborando puede restaurar el servidor copiando sus ficheros. Podemos decir que, **cada clon es realmente una copia de seguridad completa de todos los datos.**



Utilización de Git

Hay varias formas de usar git:

- A través de interfaces gráficas: <https://git-scm.com/downloads/guis>
- A través de la terminal.
 - Esta es la versión que utilizaremos en clase.



Instalación y Configuración de Git

1. Instalar:

- `$ sudo apt install git`

2. Configura tu identidad:

- `$ git config --global user.name "John Doe"`
- `$ git config --global user.email johndoe@example.com`

3. Comprobar que se ha insertado bien la identidad:

- `$ git config --list --show-origin`
- `$ cat ~/.git/config`

4. Se pueden establecer otros parámetros:

- `$ git config --global core.editor nano`

5. Más info

- `$ git help config`

Creación de repositorio local

1. `$ mkdir ~/Documents/Git/new_repo`
2. `$ cd ~/Documents/Git/new_repo`
3. `$ git init`
4. Si hacemos `$ ls -lah` veremos que se ha creado un directorio `.git/`

Git add

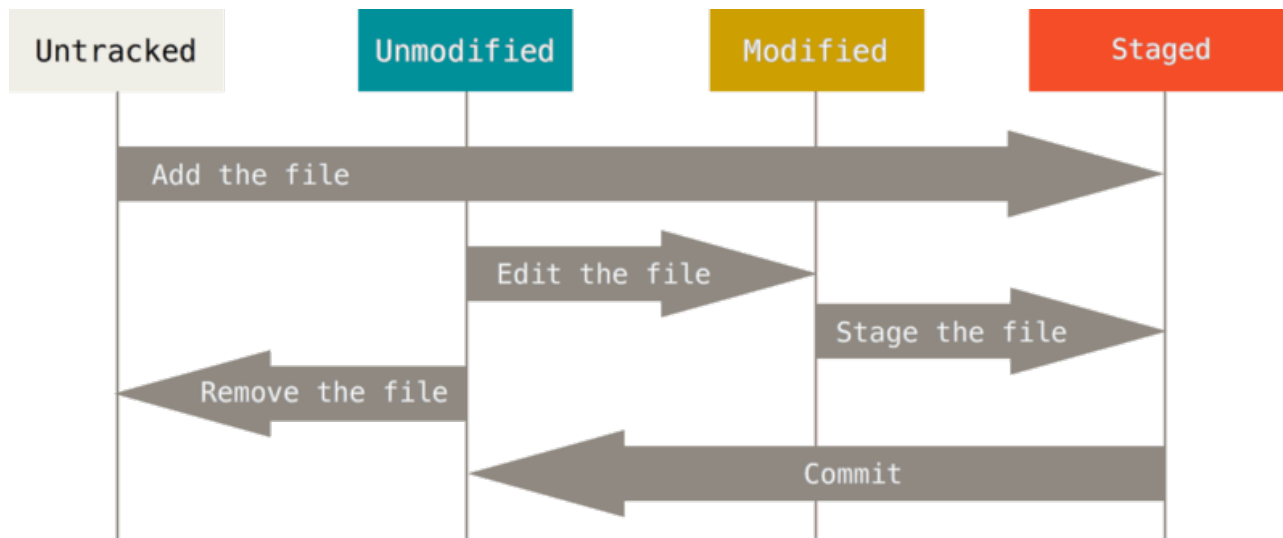
1. Descargamos del campus virtual *calculator.zip* y lo descomprimos en `~/Documents/Git/new_repo`
2. Ahora mismo tenemos ficheros en nuestro directorio, pero no están trackeados. Para ello, tenemos que añadir los ficheros. Maneras:
 - a. `$ git add *` o su equivalente `git add -a`. NO RECOMENDADO.
 - b. `$ git add <fichero1> <fichero2> ...`

Git commit

1. Para confirmar los cambios al repositorio:

- a. `$ git commit -a -m "Mensaje del commit"` NO RECOMENDADO
- b. `$ git commit -m "Mensaje del commit"` después de haber hecho `$ git add`
- c. Si no ponemos el `-m`, nos mostrará un *template* que podemos modificar.

Ciclo de vida de los fichero en Git



- Podemos comprobar en todo momento el estado de nuestro repositorio con
 - `$ git status`

Git status

- Creamos un fichero
 - `$ echo 'hola mundo' > hola`
- Comprobamos el estado de nuestro repositorio.
 - `$ git status`
- Modificamos un fichero cualquiera: `$ nano README.md`
- Comprobamos el estado de nuestro repositorio.
 - `$ git status`
- Como vemos hay cambios no rastreados y archivos sin seguimiento.
 - Archivos sin seguimiento: Add y commit.
 - Cambios sin rastrear: el fichero ya está registrado, pero no sus cambios; usamos Commit.

Git diff

- Si queremos ver que cambios tiene nuestro fichero con respecto al del repositorio.
 - `$ git diff README.md`
 - - indica lo que se ha eliminado respecto a la versión del repositorio.
 - + indica lo que se ha agregado respecto a la versión del repositorio.

```
},
{
@@ -59,10 +59,15 @@
    "Gathering past data for schizophrenia with people from medecal_city...\n",
    "Number of People with information on schizophrenia in medecal_city: 74\n",
    "Number of relevant metrics for schizophrenia: 410\n",
-   "Splitting Available data into Train: 50.00% Validation: 50.00%\n",
+   "\n",
+   "schizophrenia Patient Distribution\n",
+   "0.0: 40\n",
+   "1.0: 34\n",
+   "\n",
+   "Splitting Available data into Train: 85.00% Validation: 15.00%\n",
    "Fitting with Logistic Model...\n",
-   "Accuracy on 37 training data: 100.00%\n",
-   "Accuracy on 37 validation data: 70.27%\n",
+   "Accuracy on 62 training data: 100.00%\n",
+   "Accuracy on 12 validation data: 66.67%\n"
  ]
}
```

git checkout

Imaginemos que hemos modificado los ficheros para probar una nueva característica. ¿Qué sucede si nos damos cuenta que es inviable?

- Podemos volver a restaurar todo con:
 - `git checkout *` NO RECOMENDADO
- O bien ficheros concretos con:
 - `git checkout -- <fichero1> <fichero2> ...`
- Probad a modificar varios ficheros y observad la diferencia.

rm y git rm (con y sin --cached)

A la hora de eliminar ficheros del repositorio, no podemos confundir *rm* con *git rm*

- *rm*: Comando para borrar fichero. El fichero se mantiene en *unstaged*.
 - Habría que pasarlo a staged (*git add*) y luego *commit* para confirmar el borrado.
 - Esto se hace por seguridad.
- *git rm*: Comando de git para dejar de trackear un fichero y lo elimina.
 - Se confirma con *commit*.
- *git rm --cached*: Comando de git para dejar de trackear un fichero y NO lo elimina.
- Probad estos métodos y observad la diferencia.

mv y git mv

- *mv*: Comando para mover fichero. El fichero origen se borra y el nuevo se tiene que añadir.
 - Nuevo hay que pasarlo a staged (git add) y luego commit para confirmar el borrado del antiguo y la adicción del nuevo.
 - Esto se hace por seguridad.
- *git mv*: Comando de git para mover un fichero.
 - Se confirma con commit.
- Probad ambos métodos y observad la diferencia.

Git log

- `$ git clone https://github.com/NVIDIA/cutlass`
- `$ cd cutlass`
- Para ver el log utilizamos `git log`
 - Se mostrará en orden cronológico inverso (de más reciente a más antiguo).
- Si queremos limitar la salida:
 - `git log -p -2`
- Otros comandos útiles:
 - `git log --stat`
 - `git log -S function_name`
 - `git log --since=2.weeks`
 - `git log --oneline`
 - `git log --pretty=oneline`
 - `git log --pretty=format:"%h %s" --graph`
 - https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History#pretty_format

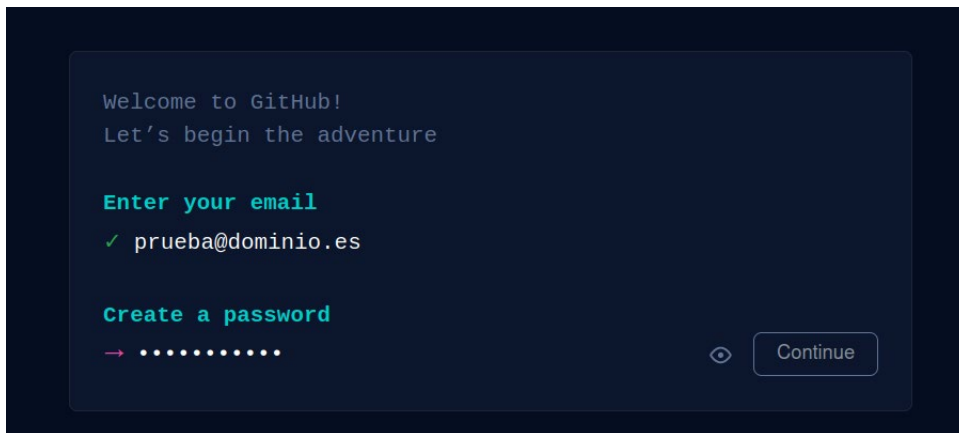
Otros comandos de git

Otros comandos de git interesantes, pero que no se explicarán en clase son:

- git reset
- git restore
- git revert

Trabajando con ramas remotas en Github

- Para trabajar con ramas en remoto vamos a utilizar Github.
- Lo primero que tenemos que tener es una cuenta.
 - Accedemos a <https://github.com/>
 - Pulsamos en sign up y rellenamos los datos.

A screenshot of the GitHub sign-up form. The background is dark blue. The text is white and light blue. It says "Welcome to GitHub!" and "Let's begin the adventure". There are two sections: "Enter your email" and "Create a password". The email field has a green checkmark and the text "prueba@dominio.es". The password field has a red arrow pointing to the first dot and a "Continue" button to the right.

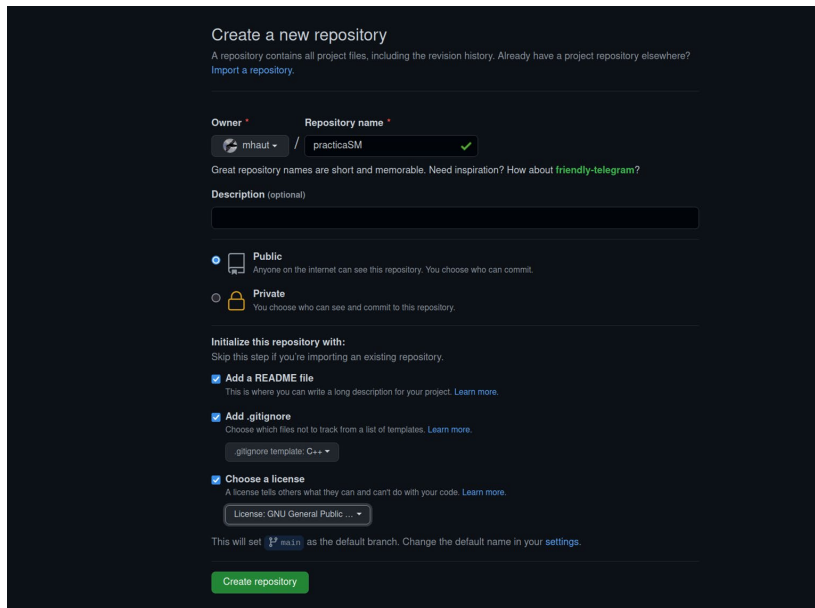
Welcome to GitHub!
Let's begin the adventure

Enter your email
✓ prueba@dominio.es

Create a password
→ Continue

Trabajando con ramas remotas en Github

- Pulsamos arriba a la derecha en el símbolo '+' y damos en new repository.
 - Configuramos el repositorio y pulsamos en *create repository*.



The screenshot shows the 'Create a new repository' page on GitHub. At the top, it says 'Create a new repository' and 'A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.' Below this, there are two dropdown menus: 'Owner' (set to 'mhaut') and 'Repository name' (set to 'practicaSM' with a green checkmark). A note says 'Great repository names are short and memorable. Need inspiration? How about friendly-telegram?'. There is a 'Description (optional)' text area. Under 'Visibility', 'Public' is selected with a radio button, and 'Private' is unselected. Below this, 'Initialize this repository with:' is shown, with a note 'Skip this step if you're importing an existing repository.' Three checkboxes are checked: 'Add a README file', 'Add .gitignore' (with a dropdown set to '.gitignore template: C++'), and 'Choose a license' (with a dropdown set to 'License: GNU General Public ...'). At the bottom, it says 'This will set `main` as the default branch. Change the default name in your settings.' and a green 'Create repository' button.

Trabajando con ramas remotas en Github

- Si todo está correcto, aparecerá el repositorio creado

The screenshot shows the GitHub interface for a repository named 'pruebaSM' by user 'mhaut'. The repository is public and has 0 stars, 1 watcher, and 0 forks. The main branch is 'main'. The repository contains three files: '.gitignore', 'LICENSE', and 'README.md', all from the initial commit. The 'README.md' file is open, showing the text 'pruebaSM'. The right sidebar contains sections for 'About', 'Releases', and 'Packages', all of which are currently empty.

mhaut / pruebaSM Public

<> Code Issues Pull requests Actions Projects Wiki Security Insights Settings

main 1 branch 0 tags

Go to file Add file Code

About

No description, website, or topics provided.

Readme 0 stars 1 watching 0 forks

Releases

No releases published
Create a new release

Packages

No packages published
Publish your first package

mhaut Initial commit fe68b1b now 1 commit

| | | |
|------------|----------------|-----|
| .gitignore | Initial commit | now |
| LICENSE | Initial commit | now |
| README.md | Initial commit | now |

README.md

pruebaSM

Git clone

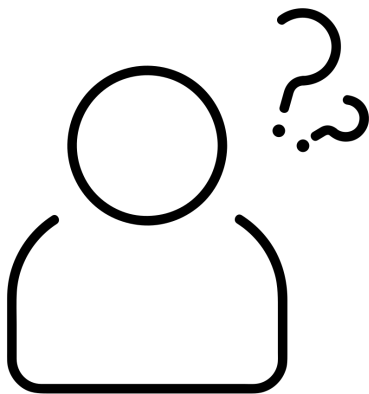
- Lo primero que tenemos que hacer es clonar el repositorio:
 - `$ git clone <url_new_repo>`
 - `$ cd <new_repo>`
 - `$ ls` (listamos los ficheros).
- **NO HACEMOS GIT INIT.** Copiamos algunos los ficheros.
- Comprobamos que ha sucedido con el comando *git status*
 - Trackeamos y confirmamos los cambios en el repositorio local
 - `git add *`
 - `git commit -a -m 'my first commit'`
- Comprobamos que ha sucedido con el comando *git status*
 - Tu rama está adelantada a 'origin/main' por 1 commit.

Git push

- Como vemos, nos indica que nuestra rama está adelantada en un commit con respecto a la remota. Sólomente si estamos seguros que nuestros cambios deben sobreescribir los remotos.
 - `git push`
- Sin embargo, al meter usuario y contraseña os da el siguiente error:
 - `remote: Support for password authentication was removed on August 13, 2021. Please use a personal access token instead.`
 - `remote: Please see https://github.blog/2020-12-15-token-authentication-requirements-for-git-operations/ for more information.`
 - `fatal: Autenticación falló para 'https://github.com/mhaut/pruebaSM/'`

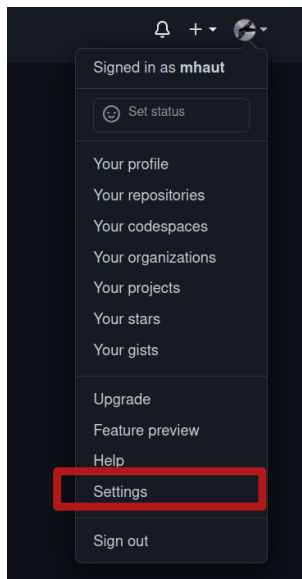
Acceder con token

- Desde Agosto de 2021, Github impide el acceso a través de usuario y contraseña. Pregunta:
 - Si esto es así, ¿por qué podemos hacer un git push?
- ¿No sería razonable que nos obligara a configurarlo desde el principio?



Creamos token de acceso

- En Github, pulsamos sobre nuestro usuario y seleccionamos la opción *settings*.



Creamos token de acceso

- En la nueva ventana, abajo a la izquierda buscamos la opción *Developer settings*.
- Configuramos el token (**DAMOS PERMISOS**) y pulsamos sobre *Generate token*.

Settings / Developer settings

GitHub Apps

OAuth Apps

Personal access tokens

Personal access tokens

Generate new token

Revoke all

Tokens you have generated that can be used to access the [GitHub API](#).

sobremesa — repo

Last used within the last week

Delete

⚠ This token has no expiration date.

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Creamos token de acceso

- En la nueva ventana os aparecerá el token, **COPIADLO** (solo tenemos una oportunidad). En caso contrario, habrá que regenerarlo.
- Configuramos el almacenamiento del token:
 - `git config --global credential.helper store`
- Con el nuevo token en el portapapeles, volvemos a hacer el push
 - `git push`
 - Introducimos el nombre de usuario.
 - Como contraseña pegaremos el token.
- Si todo se ha hecho correctamente, en el fichero:
 - `cat ~/.git-credentials` estará nuestro token.
 - `cat ~/.gitconfig` nos aparecerá **helper = store**

Git pull

- Este comando actualiza nuestro repositorio con el repositorio máster.
 - *git pull*

Ramas (branches)

- Este comando crea una nueva rama para trabajar (normalmente en local).
 - `$ git branch <branch_name>`
- Para cambiar a la nueva rama:
 - `$ git checkout <branch_name>`
- Atajo para unir los dos comandos anteriores:
 - `$ git checkout -b <branch_name>`
- Eliminar ramas:
 - `$ git branch -d <branch_name>`
- Luego podemos hacer merge de los cambios hechos en la rama desde el master:
 - `$ git merge <branch_name>`
 - Le pasaremos la opción `--no-ff` si queremos saber qué commits pertenecieron a la rama distinta del *master*.

Etiquetas (tags)

- Add, commit y luego:

- `$ git tag -a <version> -m "<tagging_message>"`

donde *version* tiene el siguiente formato: v1.4, v1.8.5, etc.

- También se puede hacer lo mismo con antiguos commits utilizando:

- `$ git tag -a <number_of_version> <commit_hash> -m "<tagging_message>"`

- `$ git push origin <number_of_version>`

.gitignore

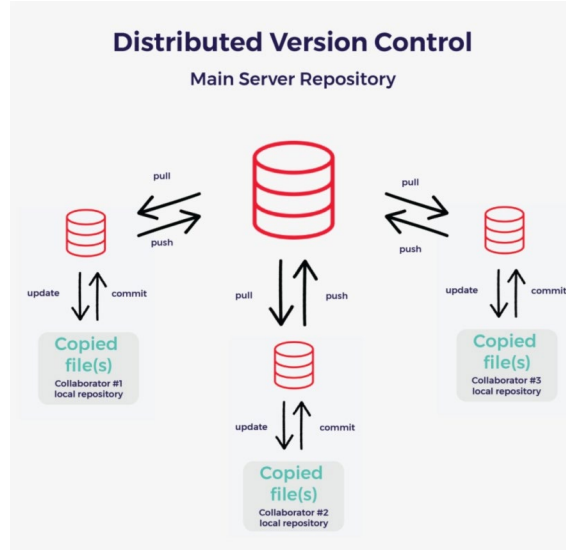
- Editando este fichero oculto podemos indicarle al repositorio qué ficheros, extensiones o carpetas no queremos trackear, es decir, no queremos que cuando hagamos push se suban al repositorio remoto.
- Añadimos un elemento en cada línea. Por ejemplo:
 - filename
 - *.out
 - folder/

Pull requests

- Los pull requests son la forma de contribuir a un proyecto grupal o de código abierto.
- Por ejemplo, un usuario llamado Manolo hace un *fork* de un repositorio que le gusta de sfandres94 para traerse el repositorio a su GitHub y le hace algunos cambios/mejoras. Ahora Manolo puede hacer un *pull request* a sfandres94, pero dependerá de este último aceptarlo. Es como decir: “sfandres94, ¿podrías por favor extraer (hacer *pull*) mis cambios ya que he hecho X mejoras?”

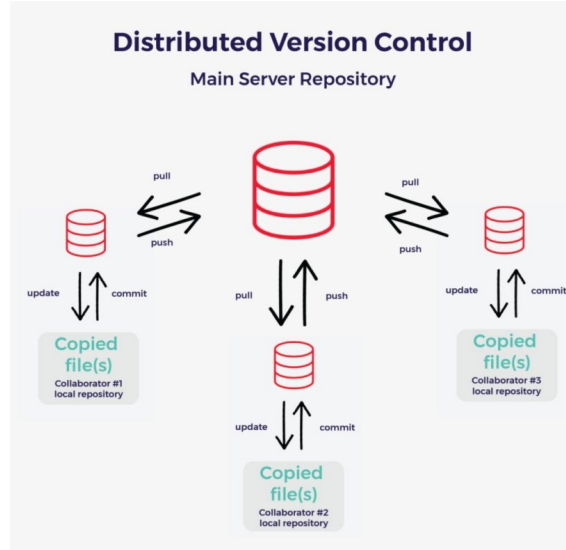
Problemas de sincronización

- Como observamos, varios usuarios pueden tener su propio repositorio y actualizar el remoto cuando consideren necesario.



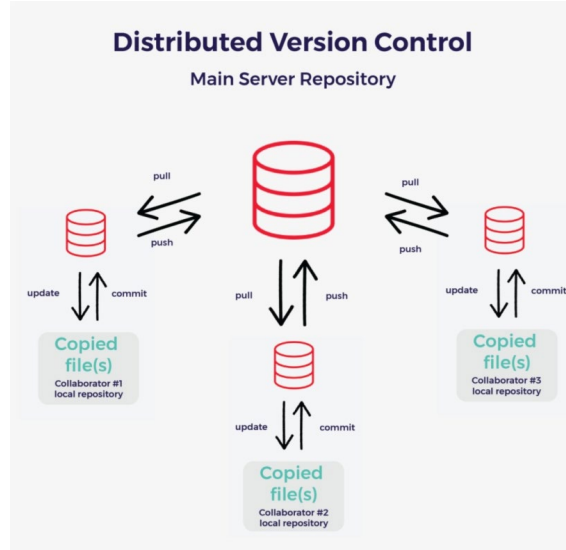
Problemas de sincronización

- ¿Qué sucede si dos usuarios hacen cambios y los confirman en su propio repositorio?



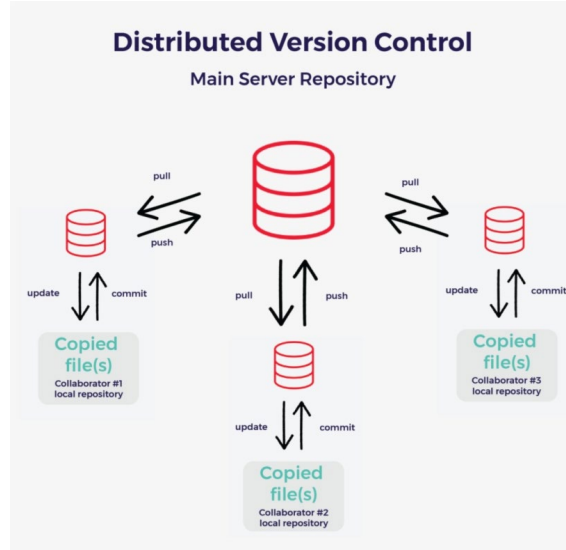
Problemas de sincronización

- ¿Podrían sincronizar con el servidor remoto?



Problemas de sincronización

- La respuesta es SÍ.
- No hay problemas si no modifican los mismos ficheros.
 - Pero, ¿y si lo hacen?



Problemas de sincronización

- Nos encontraríamos ante un conflicto que se podría resolver:
 - **Automáticamente:** Si las líneas del fichero sobre las que han estado trabajando los usuarios son diferentes.
 - **Manualmente:** En el caso de que sean las mismas.

Ejemplo:

- `cd GitHub; mkdir Repo_local_1; cd Repo_local_1; git clone <url_new_repo>; cd <new_repo>`
 - Creamos y editamos un fichero para que ponga *My first project* y confirmamos los cambios.
- `cd GitHub; mkdir Repo_local_2; cd Repo_local_2; git clone <url_new_repo>; cd <new_repo>`
 - Creamos y editamos el mismo fichero para que ponga *My second project* y confirmamos los cambios.

Problemas de sincronización

¿Por qué no hay error?

Problemas de sincronización

- Como vemos, mientras estemos trabajando con nuestro repositorio, no vamos a cometer conflictos con el repositorio remoto. Pero, ¿y si subimos cambios diferentes?

Ejemplo:

- `cd Repo_local_1/<new_repo>; git push`
 - No hay errores.
- `cd Repo_local_2/< new_repo>; git push`
 - Fijémonos en que ahora tenemos un error.

```
To https://github.com/mhaut/pruebaSM
! [rejected]        main -> main (fetch first)
error: falló el push de algunas referencias a 'https://github.com/mhaut/pruebaSM'
ayuda: Actualizaciones fueron rechazadas porque el remoto contiene trabajo que
ayuda: no existe localmente. Esto es causado usualmente por otro repositorio
ayuda: realizando push a la misma ref. Quizás quiera integrar primero los cambios
ayuda: remotos (ej. 'git pull ...') antes de volver a hacer push.
ayuda: Vea 'Notes about fast-forwards' en 'git push --help' para detalles.
greta@goodfellow:~/GITSISTEMASMULTIMEDIA/bbb/pruebaSM$
```

Problemas de sincronización

- Cuando nos encontremos con el error anterior, lo primero que tenemos que hacer es traernos el código del repositorio remoto al local:
 - `git pull`

Si lo ha integrado automáticamente aparecerá el siguiente mensaje:

```
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
Desempaquetando objetos: 100% (3/3), 267 bytes | 267.00 KiB/s, listo.
Desde https://github.com/mhaut/pruebaSM
83895f8..c0b84f1 main -> origin/main
Auto-fusionando LEAME4
CONFLICTO (contenido): Conflicto de fusión en LEAME4
Fusión automática falló; arregle los conflictos y luego realice un commit con el resultado.
greta@goodfellow:~/GITSISTEMASMULTIMEDIA/bbb/pruebaSM$
```

Problemas de sincronización

- Cuando nos encontremos con el error anterior, lo primero que tenemos que hacer es traernos el código del repositorio remoto al local:
 - `git pull`

Si lo ha integrado el código automáticamente aparecerá el siguiente mensaje:

```
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
Desempaquetando objetos: 100% (3/3), 267 bytes | 267.00 KiB/s, listo.
Desde https://github.com/mhaut/pruebaSM
  83895f8..c0b84f1  main      -> origin/main
Auto-fusionando LEAME4
CONFLICTO (contenido): Conflicto de fusión en LEAME4
Fusión automática falló; arregle los conflictos y luego realice un commit con el resultado.
greta@goodfellow:~/GITSISTEMASMULTIMEDIA/bbb/pruebaSM$
```

OJO: Integrado no es arreglado. **EL CONFLICTO SIGUE.**

Problemas de sincronización

Si abrimos el fichero, veremos que ha aparecido nuevo texto:

```
<<<<<<< HEAD
```

```
My second Project
```

```
=====
```

```
My first Project
```

```
>>>>>>> c0b84f17ef329ea3316e4bf1fb7050536572ab9e
```

Debemos elegir si lo que queremos es el contenido de nuestro repositorio (entre HEAD y ===== o el remoto entre ===== y el commit.

Problemas de sincronización

Elegiremos el más adecuado, en este caso, el del repositorio remoto y borraremos el resto:

My first Project

Finalmente debemos mandar los cambios a remoto. Observa que si confirmas cambios a tu repo local con sólo el fichero, git te dirá que no. Hay que confirmar todo, vamos a mergear:

- *git commit -a -m 'merge'*
- *git push*

Problemas de sincronización

En las ocasiones en las que un usuario ha hecho varios commits y modificado funciones que se llaman igual, la integración automática no funciona.

- Sin embargo, lo solucionaremos mediante el merge manual, para ello, utilizaremos la herramienta gráfica **meld**. Hay muchas otras *opendiff*, *kdiff3*, *tkdiff*, *xxdiff*, *meld*, *tortoisemerge*, *gvimdiff*, *diffuse*, *diffmerge*, *ecmerge*, *p4merge* *araxis*, *bc*, *codecompare*, *smerge*, *emerge*, *vimdiff*

Problemas de sincronización

Creación del conflicto

Problemas de sincronización

- `mkdir aaa; cd aaa; git clone <new_repo>; cd <new_repo>`
 - Creamos el fichero `hola_mundo.py` y lo añadimos al repositorio. Contenido:
 - `def hola_mundo():`
 - `print("Hola mundo!")`
 - `hola_mundo()`
 - Git add + commit + push
- `mkdir bbb; cd bbb; git clone <new_repo>; cd <new_repo>`
 - Actualizamos el repositorio:
 - *git pull*
 - Modificamos el fichero `hola mundo` para que tenga este contenido:
 - `def hola_mundo():`
 - `print("Hola mundo!")`
 - `def hola_mundo2():`
 - `print("Hola mundo 2!")`
 - `hola_mundo()`
 - Insertamos los cambios en el repositorio **pero no hacemos push.**

Problemas de sincronización

- Nos vamos al repositorio de la ruta aaa.
 - Modificamos el fichero con:
 - `def hola_mundo():`
 - `print("Hola mundo!")`
 - `def hola_mundo3():`
 - `print("Hola mundo 3!")`
 - `hola_mundo()`
 - Git add + commit + push
- Nos vamos al repositorio de la ruta bbb.
 - Hacemos push para mandar al remoto ---> Conflicto.

Problemas de sincronización

```
To https://github.com/mhaut/pruebaSM
! [rejected]        main -> main (fetch first)
error: falló el push de algunas referencias a 'https://github.com/mhaut/pruebaSM'
ayuda: Actualizaciones fueron rechazadas porque el remoto contiene trabajo que
ayuda: no existe localmente. Esto es causado usualmente por otro repositorio
ayuda: realizando push a la misma ref. Quizás quiera integrar primero los cambios
ayuda: remotos (ej. 'git pull ...') antes de volver a hacer push.
ayuda: Vea 'Notes about fast-forwards' en 'git push --help' para detalles.
greta@goodfellow:~/GITSISTEMASMULTIMEDIA/bbb/pruebaSM$
```

Problemas de sincronización

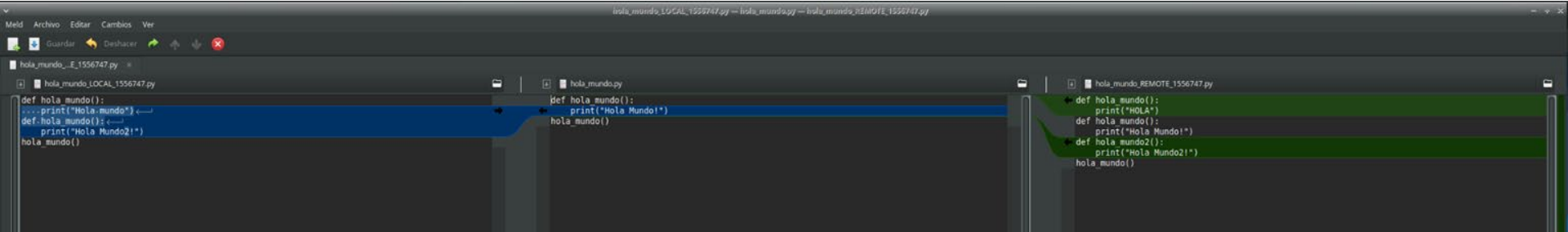
- Como observamos, tenemos un conflicto, lo primero
 - \$ git pull
- Nos indica git que no ha sido capaz de solventar el conflicto

```
greta@goodfellow:~/GITSISTEMASMULTIMEDIA/aaa/pruebaSM$ git pull
remote: Enumerating objects: 8, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 6 (delta 3), reused 6 (delta 3), pack-reused 0
Desempaquetando objetos: 100% (6/6), 535 bytes | 535.00 KiB/s, listo.
Desde https://github.com/mhaut/pruebaSM
26f7392..519e964  main      -> origin/main
Auto-fusionando hola_mundo.py
CONFLICTO (contenido): Conflicto de fusión en hola_mundo.py
Fusión automática falló; arregle los conflictos y luego realice un commit con el resultado.
```

- Como hemos comentado previamente, utilizaremos la herramienta meld.
 - \$ sudo apt install meld
 - \$ git mergetool
 - y seleccionaremos *meld* como la herramienta para solventar el conflicto.

Problemas de sincronización

- Nos aparecerán 3 ventanas (fichero local, nuestro repositorio y el remoto)



- Con ayuda de las flechas iremos pasando los códigos de un sitio a otro.
 - El conflicto estará solucionado cuando en los tres sitios haya el mismo código.



Sesión 2

Sistemas Multimedia