Curs 20-21

Unitat 3. Sistemes de control de versions

rcs, subversion, git

Entorns de desenvolupament

IES Jaume II El Just
Tavernes de la Valldigna

Continguts

1. Què són els sistemes de control de versions?

Veiem una definició del què són els sistemes de control de versions:

> Un SCVs és un sistema que registra els canvis que es produeixen en un conjunt de fitxers que formen part d'un projecte al llarg del temps, de manera que en qualsevol moment es pot tornar a una versió antiga, veure els canvis que s'han realitzat sobre determinat fitxer o qui ha fet estos canvis, entre d'altres funcionalitats.

Sovint, quan estem realitzant algun projecte o treball, tenim la necessitat de gestionar diverses versions de fitxers per tal de poder controlar els canvis que fem al llarg del temps.

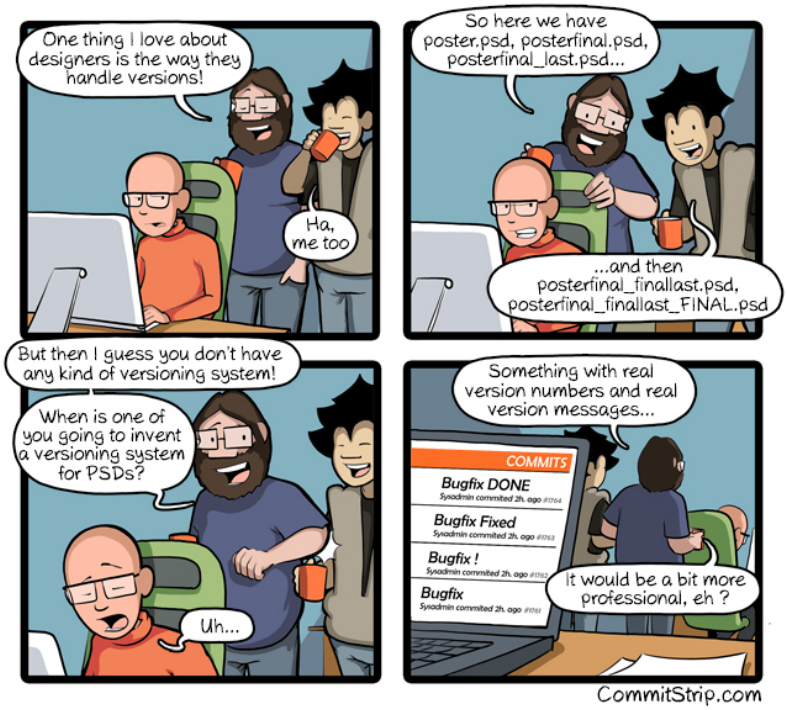

Figura 1: Control de versions

Hi ha moltes formes de portar el control de les versions, com per exemple, incloure al nom del fitxer o carpeta un número, la data en què s'ha modificat, o un text indicatiu, com veiem a la vinyeta de dalt. Aquests mètodes *manuals* de control de versions poden portar a errors, com puguen ser l'edició de versions incorrectes o la sobreescriptura d'un directori que no hauríem d'haver sobreescrit.

Quan necessitem gestionar les diferents versions dels fitxers d'un projecte, com puguen ser el codi font o la documentació, és habitual utilitzar sistemes de control de versions, en anglès *Version Control*

System (VCS) o *Source Control Management(SCM)*. Aquests sistemes ens pemetran registrar els canvis al llarg del temps, fent ús d'***instantànies*** del nostre projecte, anomenades ***commits***, que s'aniran emmagatzemant a la base de dades del sistema de control de versions. Amb aquesta informació guardada, el sistema ens permetrà:

- Revertir (tornar) un o diversos fitxers a un estat previ (instantània prèvia),
- Comparar els canvis produits en determinats fitxers,
- Veure qui ha realitzat determinat canvi al codi.

Els sistemes de control de versions ens són útils per gestionar diferents versions de fitxers de qualsevol tipus, però ofereixen tota la seua potència quan es tracta de fitxers de text pla, com puguen ser els fitxers font, amb llenguatges de marques, o fitxers de documentació Markdown, davant altres formats binaris. El motiu és que l'eina que utilitzen aquests sistemes per sota es basa en la comparació de fitxers línia a línia.

Veiem-ho amb un exemple. Disposem de dos fitxers de documentació Markdown: `projecte1.md` i `projecte2.md`:

```
1 # Projecte EDD
2
3 Aquesta és la primera versió del document del projecte EDD.
```

```
1 # Projecte EDD
2
3 Aquesta és la segona versió del document del projecte EDD.
4
5 A mes de la linia de dalt modificada, afegim una linia nova.
```

Fem ús de l'ordre `diff` per vore les diferències:

```
1 $ diff projecte1.md projecte2.md
2 3c3,5
3 < Aquesta és la primera versió del document del projecte EDD.
4 ---
5 > Aquesta és la segona versió del document del projecte EDD.
6 >
7 > A mes de la linia de dalt modificada, afegim una linia nova.
```

Com veiem, ens mostra que hi ha diferències, concretament que la línia 3 ha canviat (`3c3`), i quin ha setat el canvi,i a més, que hi ha una línia nova a la línia 5.

Per altra banda, si el que fem és comparar fitxers binaris, per exemple, dos fitxers comprimits amb `tar.gz`, obtindrem:

```
1 $ diff Fitxer1.tar.gz Fitxer2.tar.gz
2 Binary files Fitxer1.tar.gz and Fitxer2.tar.gz differ
```

És a dir, ens indica que hi ha canvis, però no quins canvis s'han realitzat.

2. Tipus de Sistemes de control de versions

Els sistemes de control de versions tenen un llarg recorregut, i els primers en aparéixer ho van fer per la dècada dels 70 del segle passat. Amb el temps, han anat guanyant complexitat i funcionalitat. En aquest apartat, veurem una classificació dels diferents tipus de sistemes de control de versions que ens podem trobar.

2.1. Sistemes de control de versions local

Són els primers sistemes que van aparéixer, i el seu funcionament és bastant senzill: Per a les diferents versions es van guardant les diferències versions entre els fitxers, de manera que per obtindre una versió qualsevol, s'apliquen les diferències sobre el fitxer base.

Gràficament, podriem descriure aquests tipus de sistemes d'aquesta manera:

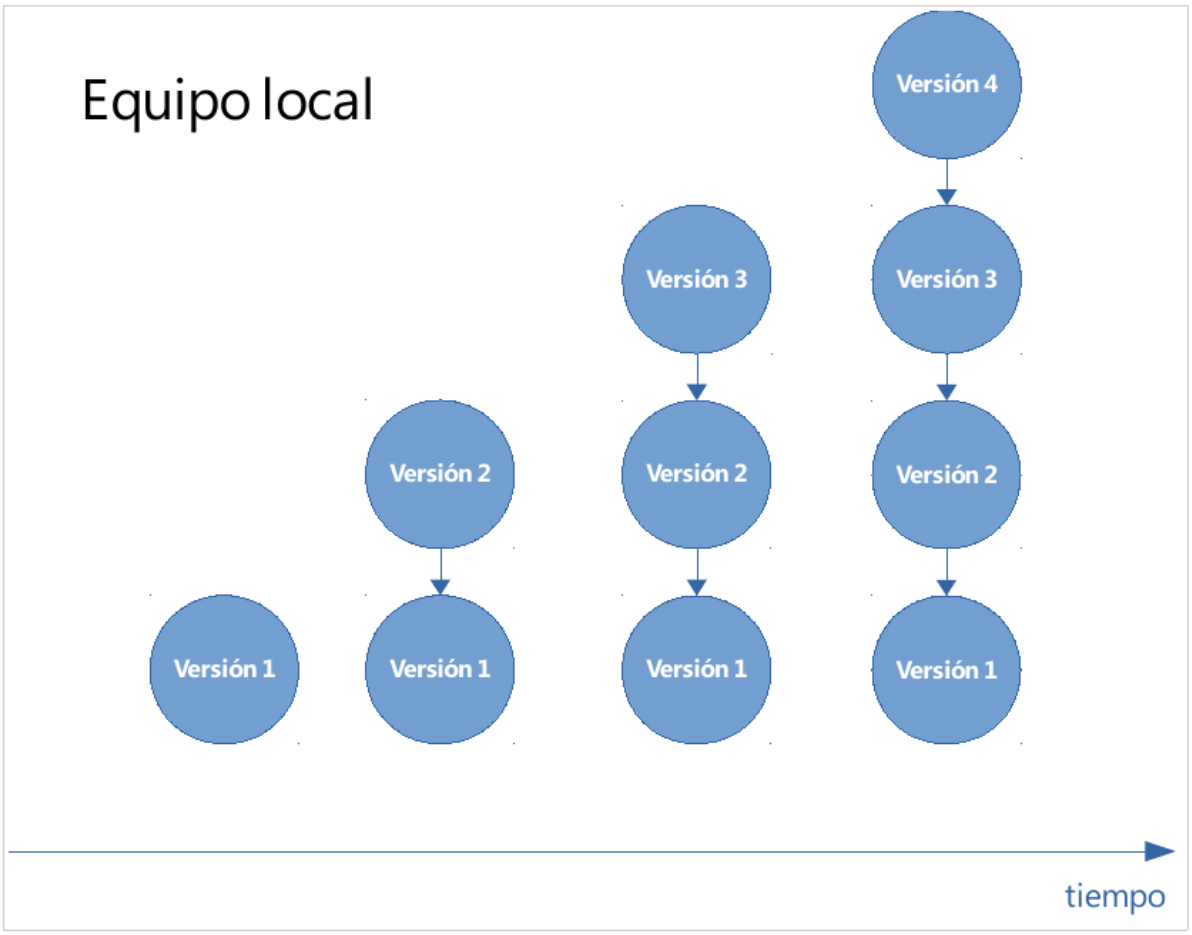

Figura 2: SCV local

El primer sistema d'aquest tipus que va aparéixer va ser *SCCS*, l'any 1972, amb llicència propietària, i va tenir bastant èxit. Posteriorment, el projecte *GNU* va llançar el seu reemplaç lliure: *CSSC (Compatibly Stupid Source Control)*, disponible actualment als sistemes GNU/Linux, i posteriorment RCS.

Veiem un exemple sobre com utilitzaríem aquest sistema de control de versions.

Exemple pràctic de SCV local: RCS

En primer lloc, caldrà instal·lar els paquets `rcs` i `rcs-blame`:

```
1 sudo apt-get install rcs rcs-blame
```

Una vegada descarregats, podem fer ús d'ells per tal de portar a terme el control de versions.

Anem a crear, dins un directori de treball un fitxer anomenat `fitxer.txt`, amb el següent contingut:

```
Fitxer, versió 1
```

El guardem i eixim. Si feu un ls -l veureu que disposem els permisos habituals (rw-rw-r–) sobre el fitxer:

```
-rw-rw-r-- 1 alumne alumne    0 'doct 18 06:23 fitxer.txt
```

El primer que haurem de fer per tal de portar un control de versions sobre aquest fitxer serà ***registrar-lo (check in)*** al sistema de control de versions. Per a això farem:

```
$ ci -u fitxer.txt
fitxer.txt,v  <--  fitxer.txt
enter description, terminated with single '.' or end of file:
NOTE: This is NOT the log message!
>> v1
>> .
initial revision: 1.1
done
```

Com veiem, hem fet ús de l'ordre `ci`, que significa *check-in*, i ens serveix exactament per això, per indicar que el fitxer va a estar sota el control de versions.

I com sap RCS que el fitxer ara està sota el control de versions? Si tornem a fer un `ls -l` sobre el directori de treball, ens trobem:

```
$ ls -l
total 8
-r-xr-xr-x 1 alumne alumne  18 oct 18 06:23 fitxer.txt
-r-xr-xr-x 1 alumne alumne 200 oct 18 06:25 fitxer.txt,v
```

Fixeu-vos en dos detalls:

- El fitxer original ara ja no té permís de lectura
- S'ha creat un fitxer nou, anomenat fitxer.txt,v, amb els mateixos permisos que té ara l'original.

Fixeu-vos que quan hem fet el `check-in`, ens indica la generació d'un fitxer de control de versions `fitxer.txt,v` i a més ens ha demanat posar una descripció. **Recordeu que per finalitzar d'escriiure la descripció hem d'escriure una línia amb un únic punt**. Aquesta descripció s'inclourà també dins el fitxer de control de versions.

Bé, vist açò, veiem què fa RCS per marcar el control de versions. Quan registrem un fitxer en RCS, aquest pot fer dues coses per recordar a l'usuari que el fitxer es troba sota el control de RCS:

- Elimina el fitxer original, deixant només el fitxer sota control de RCS. Aquest nou fitxer sota el control de versions s'anomena per defecte com l'original, i s'acaba amb `,v`. Generalment, es

conservarà al mateix directori que l'original, sempre i quan no tinguem creada una carpeta en el mateix directori anomenada `RCS` (creada per l'usuari); en eixe cas, es guardaria en esta carpeta.
- A més, si fem ús de l'opció `-u`, torna a revisar el fitxer (no l'esborra), però deixa els seus permissos com a només lectura.

Aleshores, *com modifiquem un fitxer sota control de versions si és només de lectura?* Per a això hem de fer un *check out* del fitxer, amb:

```
$ co -l ./fitxer.txt
./fitxer.txt,v  -->  ./fitxer.txt
revision 1.1 (locked)
done
```

Fixeu-se ara que al fer el `co` ens ha indicat que tenim un nou `fitxer.txt` a partir del fitxer `fitxer.txt,v`, i si tornem a llistar la carpeta:

```
$ ls -l
total 8
-rwxr-xr-x 1 alumne alumne  18 oct 18 06:55 fitxer.txt
-r-xr-xr-x 1 alumne alumne 212 oct 18 06:55 fitxer.txt,v
```

Veurem com disposem de permís de lectura sobre el `fitxer.txt`. L'opció `-l` (*lock*) de `co` ens serveix per *bloquejar* el fitxer, de manera que altre usuari del sistema no el puga modificar mentre estem fent els canvis.

Si fem ara un cop d'ull al fitxer de control de versions podreu deduïr alguna informació d'interès, com les diferents revisions que s'han anat fent, o quin usuari té el fitxer bloquejat.

Una vegada fet el checkout, podem modificar el fitxer de nou, per exemple amb el següent contingut:

```
Fitxer, versió 1

Afegim més text al fitxer, per veure com ho gestiona el RCS.
```

Com veiem, hem afegit una línia més. Ara per indicar a RCS que hem acabat de fer canvis per a aquesta nova versió, i que desbloquege el fitxer. Per a això, farem de nou un `check-in`:

```
$ ci -u ./fitxer.txt
./fitxer.txt,v  <--  ./fitxer.txt
new revision: 1.2; previous revision: 1.1
enter log message, terminated with single '.' or end of file:
>> Segona versió del fitxer
>> .
done
```

Amb el que ja tindrem de nou el fitxer sense permís d'escriptura. Si donem un cop d'ull al `fitxer`

`.txt,v`, veurem com han aparegut més canvis. Aquest fitxer, anirà guardant de forma eficient les modificacions cada vegada que es fa el registre o *check-in* del fitxer, i té un **número de revisió**, que guarda junt als comentaris que afegim a cadascuna.

A partir d'aquesta informació, disposem d'eines per tal de revisar els canvis realitzats. Per fer això, disposem de l'ordre `rlog`. Veiem el resultat d'aplicar-la sobre el nostre fitxer:

```
$ rlog ./fitxer.txt

RCS file: ./fitxer.txt,v
Working file: ./fitxer.txt
head: 1.2
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 2; selected revisions: 2
description:
v1
----------------------------
revision 1.2
date: 2019/10/18 05:11:05;  author: alumne;  state: Exp;  lines: +2 -0
Segona versió del fitxer
----------------------------
revision 1.1
date: 2019/10/18 04:24:55;  author: alumne;  state: Exp;
Initial revision
=============================================================================
```

Ara, si volem veure l'estat d'una versió concreta, farem ús de `co -p x.y fitxer`, sent *x.y* el número de versió. Per exemple, per veure les dues revisions del fitxer, tenim:

```
$ co -p1.1 ./fitxer.txt
./fitxer.txt,v  -->  standard output
revision 1.1
Fitxer, versió 1
```

```
$ co -p1.2 ./fitxer.txt
./fitxer.txt,v  -->  standard output
revision 1.2
Fitxer, versió 1

Afegim més text al fitxer, per veure com ho gestiona el RCS.
```

Si ara el que volem és veure les diferències entre les dues versions, fem ús de `rcsdiff`.

```
$ rcsdiff -r1.1 -r 1.2 ./fitxer.txt
rcsdiff: RCS/1.2,v: No such file or directory
```

```
3  ===================================================================
4  RCS file: ./fitxer.txt,v
5  retrieving revision 1.1
6  retrieving revision 1.2
7  diff -r1.1 -r1.2
8  1a2,3
9  >
10 > Afegim més text al fitxer, per veure com ho gestiona el RCS.
```

Com veiem, el que fa internament és utilitzar un `diff` entre les dues versions del fitxer.

Finalment, si el que volem és tornar a una versió anterior del fitxer, només haurem de fer un *check out* amb el número de revisió concret:

Tenint la versió 1.2 del fitxer:

```
1 $ cat fitxer.txt
2 Fitxer, versió 1
3
4 Afegim més text al fitxer, per veure com ho gestiona el RCS.
```

Revertim a la versió 1.1:

```
1 $ co -r1.1 ./fitxer.txt
2 ./fitxer.txt,v  -->  ./fitxer.txt
3 revision 1.1
4 done
5
6 $ cat fitxer.txt
7 Fitxer, versió 1
```

Exercici. Realitzeu les següents operacions amb rcs:

1. Creeu un fitxer anomenat `exercici.md`, i afegiu-li algun contingut en format Markdown.
2. Registreu el fitxer per a que el controle RCS.
3. Intenteu modificar el fitxer, per veure si teniu o no permís.
4. Feu un *checkout* del fitxer, ara sí, per afegir canvis.
5. Modifiqueu el fitxer i afegiu-li més contingut. Aquesta serà la versió 1.2.
6. Amb un altre usuari, intenteu modificar el fitxer (fent el checkout abans).
7. Com a l'usuari original, registreu els canvis al fitxer.
8. Amb l'altre usuari de nou, intenteu ara modificar el fitxer (també fent un checkout).
9. Si podeu, afegiu més informació al fitxer, ara serà la versió 1.3.
10. Com a l'usuari original, visualitzeu les diferències entre les versions, i torneu a la versió 1.2.
11. Apliqueu més canvis i guardeu-los amb la versió 1.4.

2.2. Sistemes de control de versions centralitzats

Amb l'arribada d'Internet, sorgeix la possibilitat de què diversos programadors puguen treballar de forma conjunta en un projecte, i portar-ne el control de versions amb els canvis que fa cadascun. Apareix doncs la necessitat de centralitzar el control de versions en un únic servidor que emmagatzeme les diferents versions dels fitxers.

En aquests tipus de sistemes de control, els programadors que desitgen treballar en un projecte, faran ús d'una eina client amb la qual descarregaran el projecte des del servidor, i quan acaben de realitzar canvis, els enviaran al servidor.

Existeixen diversos sistemes de control de versions, com *Concurrent Versions System (CVS, 1986)*, *Microsoft Visual SourceSafe (VSS, 1994) o amb el que treballarem aquest apartat: Subversion (SVN, 2000).

El funcionament general d'aquest tipus de sistemes de control de versions és el que es mostra a la imatge següent:

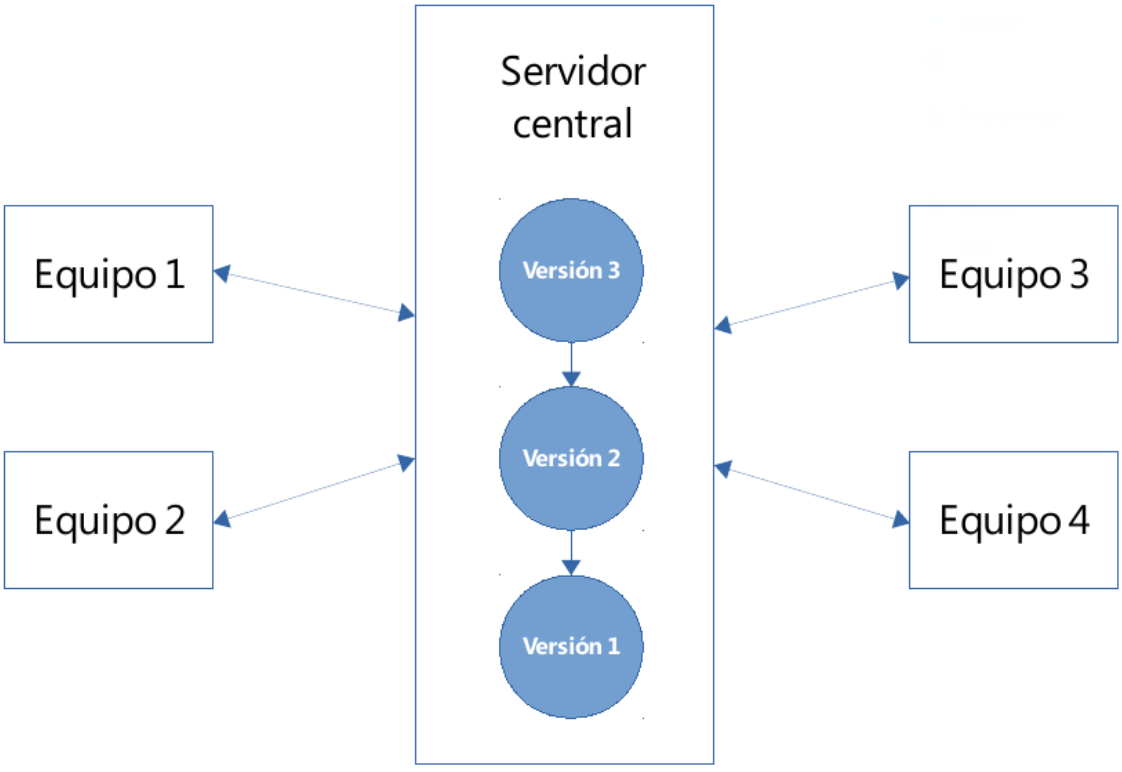

Figura 3: SCV Centralitzat

Veiem com hi ha un servidor que disposa de tres versions d'un conjunt de fitxers (anomenat dipòsit o

repositori). Els clients obtenen la última versió dels fitxers, treballen localment, i actualitzen els canvis al servidor.

Amb aquesta arquitectura guanyem versatilitat, però al estar tot centralitzat en un únic servidor, en cas que aquest falle, no podem compartir les modificacions del codi. Per altra banda, si perdem la informació del servidor, perdrem també tot l'historial de versions.

Exemple pràctic de SCV centralitzat: Subversion

Al document annexe sobre Docker, hem vist com instal·lar el servidor *Subversion Edge* en un contenidor. Per als exemple següents, anem a donar per fet que tenim el servidor funcionant en `http://localhost:18080/svn/`.

Si accedim a la URL `http://localhost:18080/viewvc` a través d'un navegador web, podrem navegar a través d'aquest pels diferents repositoris que tenim creats al servidor:

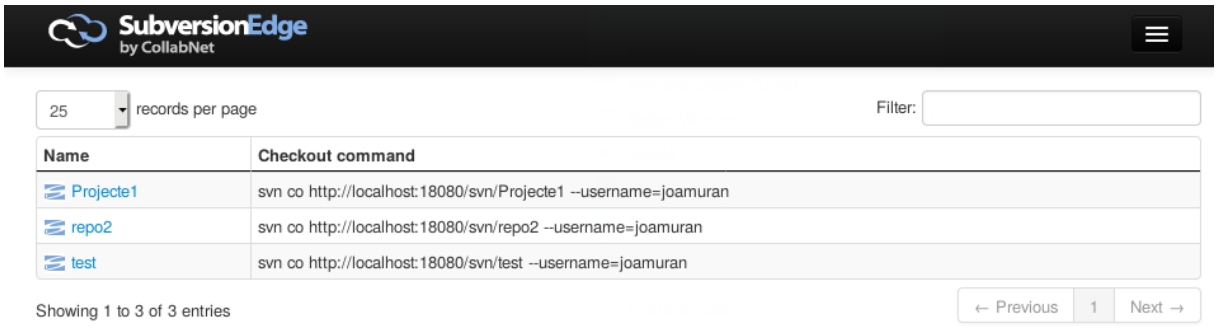

Figura 4: Navegació web pel servidor svn

En aquesta finestra se'ns mostra l'adreça amb la que podem accedir a través d'un client de Subversion a aquest repositori.

Instal·lació de subversion (client) El paquet que ens ofereix el client de subversion, s'anomena `subversion`, i podem instal·lar-lo de la manera habitual amb apt:

```
1 $ sudo apt install subversion
```

Aquest paquet ens proporciona una sèrie d'ordres `svn *` per tal de treballar contra el servidor de Subversion.

Descarregant un projecte del servidor Si no ho heu fet encara, creeu un repositori nou al servidor, anomenat, per exemple *Projecte1*. Com veurem al navegador, aquest ens suggereix directament

l'ordre *Checkout Command* com a `svn co http://localhost:18080/svn/Projecte1 --username=joamuran`.

Com viem, amb `svn co` fem un *check out* del repositori. És a dir, descarreguem la última versió d'aquest. El fet de posar darrere el nom de l'usuari és opcional, i si no el posem, agafarà per defecte el nostre nom d'usuari local. En cas que no ens autentiquem a la primera, ens demanarà un nom d'usuari nou.

Així doncs, si llancem el checkout:

```
$ svn co http://localhost:18080/svn/Projecte1 --username=joamuran
Authentication realm: <http://localhost:18080> CollabNet Subversion
   Repository
Password for 'joamuran': *********
A    Projecte1/trunk
A    Projecte1/branches
A    Projecte1/tags
Checked out revision 1.
```

Com veiem, ens indica que ha creat el directori `Projecte1` amb tots el contingut que per defecte tenía, i ens indica el número de revisió de què es tracta. Fixeu-se que davant de cada directori ens mostra una `A`, indicant que s'ha afegit (Add) el fitxer.

Ara, si entrem dins la carpeta `Projecte1`, podrem anar treballant per pujar els canvis posteriorment.

Si recordem, en generar el projecte en *Subversion Edge*, ens ha demanat si volíem generar l'estructura estàndard *trunk/branches/tags* per al projecte. Al cas de l'exemple hem seleccionat que sí que es creara, i com veiem, es tracta de les carpetes que acabem de descarregar.

Aquesta estructura és la recomanada per als projectes amb Subversion, i a la qual tenim les carpetes:

- `trunk`, amb la *línia principal* o *troncal* de desenvolupament.
- `branches`: On tenim diferents *branques* de desenvolupament, és a dir, bifurcacions de la branca troncal, per exemple per anar afegint noves funcionalitats a una aplicació. Quan el desenvolupament d'una branca es vol incorporar a la branca *principal*, es mesclen els canvis, amb el que s'anomena una operació *merge*.
- `tags`: Els tags representen com *snapshots* o *instantànies* del projecte en un moment donat. Pot tractar-se d'instantànies que contenen les versions estables d'na aplicació, per exemple.

De moment, anem a treballar únicament amb la branca principal `trunk`, on crearem un fitxer anomenat `exemple1.js`, amb el següent codi:

```
function saluda(){
    console.log("Hola Subversion");
}
```

```
4
5 saluda()
```

Com veiem, es tracta de codi *javascript* que podrem executar des de consola amb `nodejs`:

```
1 $ nodejs trunk/saluda.js
2 Hola Subversion
```

Si ara, des de dins el directori on estem (Projecte1) llancem l'ordre `svn st` (status), obtindrem:

```
1 $ svn st
2 ?       trunk/saluda.js
```

fixeu-se que ens indica un signe d'interrogant `?` i la ruta al fitxer `trunk/saluda.js`. Açò ens indica que aquest fitxer és nou i **no està sota el control de versions**.

Per afegir el fitxer al control de versions, fem ús de l'ordre `svn add`, de la següent forma:

```
1 $ svn add trunk/saluda.js
2 A         trunk/saluda.js
```

Com veiem, ara, en lloc d'un `?` ens mostra una `A`, que indica que **hem afegit aquest fitxer al control de versions, però encara no l'hem pujat al servidor**.

Si fem de nou un `svn st`, veurem que ens indica la mateixa línia. Per tal de pujar els canvis al servidor, haurem de fer un *commit*, amb `svn ci`, de la següent manera:

```
1 $ svn ci -m "Afegint el fitxer saluda.js"
2 Authentication realm: <http://localhost:18080> CollabNet Subversion
     Repository
3 Password for 'joamuran': *********
4
5 Adding         trunk/saluda.js
6 Transmitting file data .done
7 Committing transaction...
8 Committed revision 2.
```

Com veiem, ara ens indica que està afegint el fitxer, i que la versió resultant és la 2. Si ara naveguem a través del repositori, veurem que el fitxer ja està pujat, i si fem un `svn st`, veurem que ja no ens mostra cap missatge, indicant que no ens queda res per pujar al servidor.

Per altra banda, com que ja tenim el repositori configurat (a la carpeta `Projecte1` hi ha una subcarpeta `.svn` amb la informació sobre el repositori remot), per tal d'actualitzar la nostra carpeta per si altre usuari ha pujat canvis, només haurem de fer un *update* amb `svn up`. Sempre serà una bona pràctica fer aquesta operació abans de posar-nos a fer qualsevol modificació.

```
1 $ svn up
2 Updating '.':
```

```
Authentication realm: <http://localhost:18080> CollabNet Subversion
   Repository
Password for 'joamuran': *********

A    trunk/prova.txt
Updated to revision 4.
```

Veiem un xicotet exercici:

- Com a altre usuari, fes un checkout del projecte que acabem de crear (evidentment, en altre directori).
- Crea dins la carpeta trunk un altre fitxer, amb el contingut que desitges, afig-lo al contorl de versions, i puja'l al servidor.
- Com a l'usuari original, crea altre fitxer al directori trunk, afig-lo al control de versions i intenta pujar-lo al servidor. T'ha donat algun problema? I si el fitxer que anem a pujar es diu igual que el que ha pujat l'usuari anterior?

Referències sobre SVN:

- Llibre de Subversion de Collab.net: https://www.open.collab.net/community/subversion/svnbook
- Tutorial de subversion a la wiki de Debian: https://wiki.debian.org/SVNTutorial

2.3. Sistemes de control de versions distribuits

Els sistemes de control de versions distribuïts adopten un enfocament més semblant a un peer-to-peer que a una arquitectura client-servidor. En lloc de disposar d'un únic repositori centralitzat i amb el que se sincronitzen els clients, cada equip té una còpia completa de tot el repositori, en lloc de la última instantània del projecte. Així, en cas que hi haja algun problema amb el servidor central, que s'ha establert per conveni, aquest podrà ser regenerat a partir de qualsevol altre client.

Veiem-ho de forma gràfica:

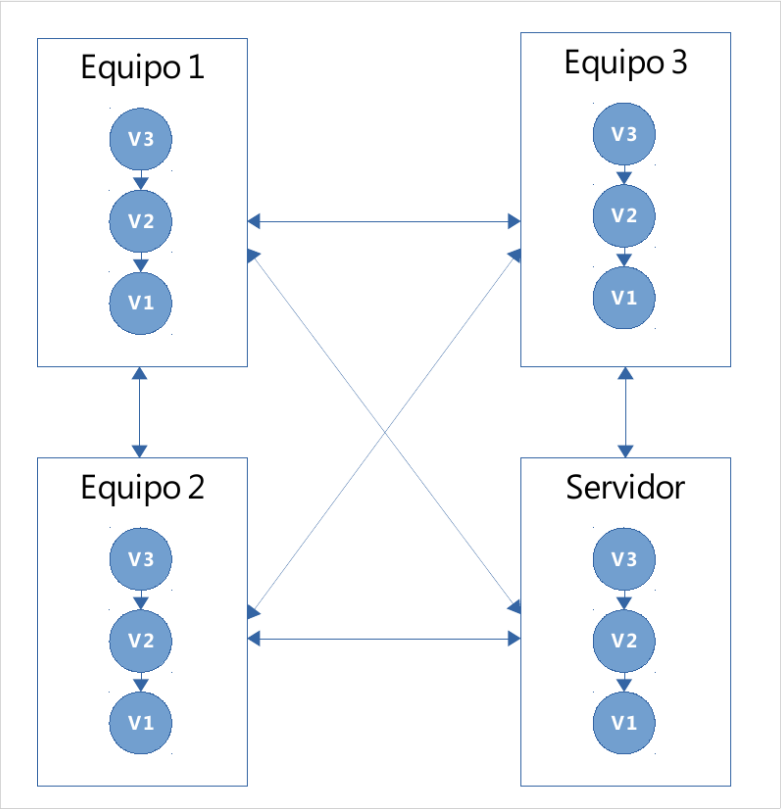

Figura 5: Control de versions distribuït

Veiem com el servidor es comporta com un host més. Cada equip disposa d'una còpia del conjunt de versions del projecte, de manera que puguen treballar de forma local, sense necessitat de connectar-se a altres equips, llevat de quan hàgen de compartir el treball o obtenir el treball d'altres companys.

Els SCV Distribuits més coneguts són BitKeeper (1998), Bazaar (2005), Mercurial (2005) i Git (2015).

3. Git

En aquest apartat, veurem en profunditat el sistema de control de versions distribuït *Git*, un dels sistemes més utilitzats en l'actualitat, i amb el que treballa la majoria de repositoris púbilcs de programari (Github, Gitlab…).

3.1. Introducció a git

Git va aparéixer per tal de donar suport al desenvolupament distribuït del kernel de Linux. Les seues principals característiques són:

- És **ràpid i escalable**, que s'adapta tant a projectes individuals com a grans projectes on col·labora una gran quantitat de desenvolupadors,
- Utilitza una **còpia completa** amb tot l'historial de desenvolupament en cada usuari local. En cas que el servidor principal deixe d'estar operatiu, es pot recuperar a partir de qualsevol cpòpia local. A més, ens permet tornar a un estat previ (commit) del projecte o veure les diferències entre dos estats, sense necessitat d'equips externs. Presenta un **desenvolupament distribuÑit**, on es disposa d'una còpia completa del repositori que facilita el desenvolupament simultani i independent mitjançant repositoris privats, evitant així la sincronització contínua.
- Permet el **treball local**, sense connexió a altres xarxes, ja l'historial de *commits* es guarda localment, pel que comprovar canvis entre versions es pot fer de forma local. Així, només ens hem de connectar a l'exterior quan volem col·laborar amb altres usuaris i obtindre els canvis des de la última connexió.
- Permet treballar amb **branques** i fussionar-les fàcilment, amb el que aconseguim un desenvolupament no linial.
- En lloc de treballar amb llistes de canvis amb les modificacions, *git* fa ús d'***instantànies*** del sistema de fitxers en cada *commit*.
- Permet múltiples **protocols**: HTTP, HTTPS, SSH, a banda del protocol propi.
- És un sistema **robust**, ja que utilitza internament un algrisme de xifrat SHA-1 de manera que qualsevol dada modificada siga detectada per *Git*.
- És un sistema **lliure**, seguint amb la filosofía de la majoria de projectes col·labortius, i es distribueix amb llicència GNU-GPL 2.0.

3.2. Instal·lació

Per tal d'instal·lar `git` en un sistema Ubuntu, ho fem a través d'apt:

```
1 sudo apt update
2 sudo apt install git
```

Per saber la versió que tenim instal·lada fem ús de `git --version`.

Per a sistemes Windows podem consultar les webs https://gitforwindows.org/ i per a Mac i Windows (i també Linux), des la web (https://git-scm.com/)[https://git-scm.com/]. Disposeu de més informació per a la instal·lació en Windows al document *Fundamentos de git*, de Jesús Amieiro Becerra.

3.3. Primeres passes amb git

3.3.1. Ajuda en git

Si escrivim des de consola l'ordre `git` sense arguments, obtindre una ajuda amb els paràmetres que aquest suporta, i les ordres més utilitzades, junt amb una xicoteta explicació.

Per altra banda, si volem ajuda d'una ordre concreta de git, farem `git help ordre`, amb l'*odre* de la que volguem treure'n lajuda.

Per altra banda, també podem consultar les pàgines del manual de Git en entorns Linux (`man git`).

3.3.2. Configuració inicial

Després d'instal·lar *Git*, i abans d'utilitzar-lo, hem de configurar alguns paràmetres. Açò ho farem amb `git config`, que permet guardar la configuració a nivell de sistema, d'usuari o de projecte.

Anem a treballar a **nivell d'usuari**, de manera que la configuració es guardarà al fitxer `.gitconfig` del nostre home d'usuari. En general, per establir un paràmetre a nivell d'usuari, ho farem amb `git config --global paràmetre valor`, mentre que els podrem consultar amb `git config --global paràmetre`.

Veiem a continguació els diferents paràmetres que cal configurar a *git* i com fer-ho:

- **Identitat de l'usuari**, que es compon del nom i el correu electrònic:

```
$ git config --global user.name "José"
$ git config --global user.email jmurcia@ieseljust.com
```

- **Editor per defecte,** per a quan *git* necessita que escriguem algun missatge (per exemple el missatge de cada commit)

```
joamuran@lluc:~$ git config --global core.editor vim
```

També existeixen altres elements a configurar que ens poden ser d'utilitat:

- Per a que utilitze l'eixida en colors significatius:

```
$ git config --global color.ui true
```

- Per indicar a git que realitze conversions entre finals de línia quan treballem en entorns híbrids Linux/Windows/Mac:

```
git config --global core.autocrlf true
```

Finalment, si volem consultar tota la llista de paràmetres, farem:

```
1 git config --list
```

3.4. Conceptes sobre git

Abans de seguir, anem a fer un repàs de conceptes que utilitza *git*. Alguns d'ells ja ens sonaran d'altres sistemes de control de versions, i d'altres ens seran nous:

- **Repositori** o dipòsit: Contenidor o base de dades on es guarda l'històric de canvis en els fitxers del projecte, guardats en ell mitjançant un *commit* o confirmació.
- **Commit** o confirmació: Acció amb la qual guardem al repositori una instantània del projecte, junt amb informació sobre aquest canvi, generalment l'autor, la data i l'explicació dels canvis.
- **Zones en git**: Un fitxer pot estar localment en tres zones:
  - ***Directori de treball*** o *working directory* és la zona del repositori visible (la veiem amb un `ls -la` o `dir /p`) i sobre la que treballarem habitualment.
  - ***Zona de preparació***, *stagging area* o *index* és una zona intermèdia, no accessible des de la terminal, però si amb git, on es troben les instantànies dels fitxers que es guardaran en el repositori en el proper *commit*.
  - El ***repositori***, pròpiament dit, on es guarden els canvis després d'haver fet els *commits*.

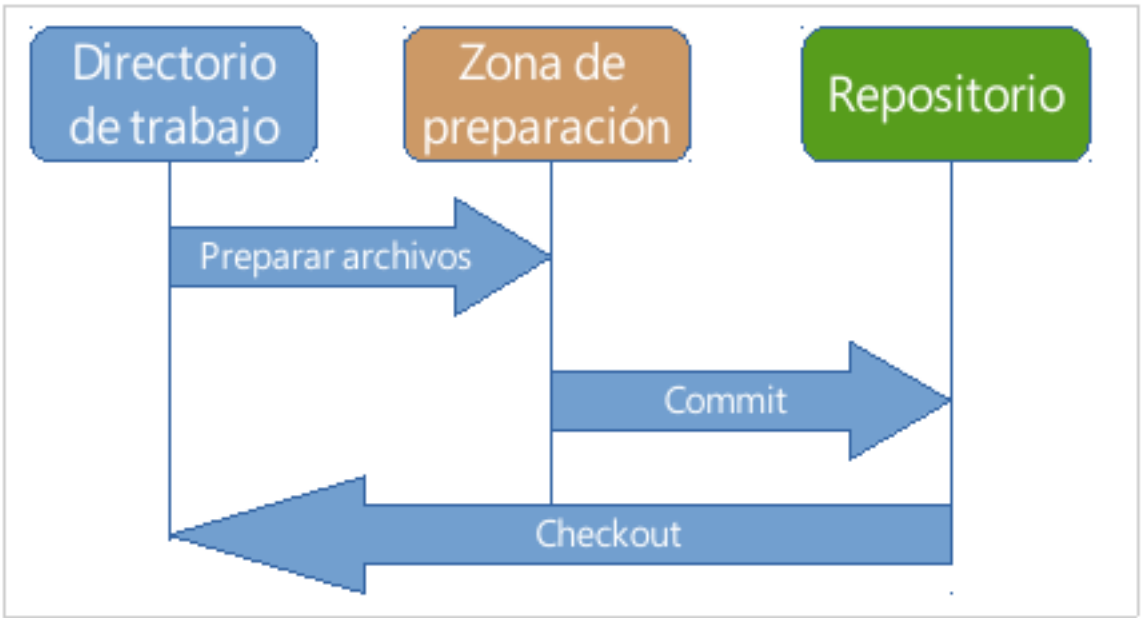

Figura 6: Zones Git

- **Estats d'un fitxer**: Els fitxers al directori de treball poden estar **sense seguiment** o **baix seguiment** de git. Els fitxers que tenim *baix seguiment*, poden presentar els següents estats:
  - ***Sense modificacions***, on el contingut del fitxer a la zona de treball, de preparació i repositori és el mateix.
  - ***Modificat***, on el contingut de la zona de treball és diferent al de la zona de preparació i al repositori.
  - ***Preparat***, on el contingut del fitxer en la zona de treball i de preparació és el mateix, però difereix del repositori, i per tant, necessitarà un *commit* en algun moment per sincronitzar-se.

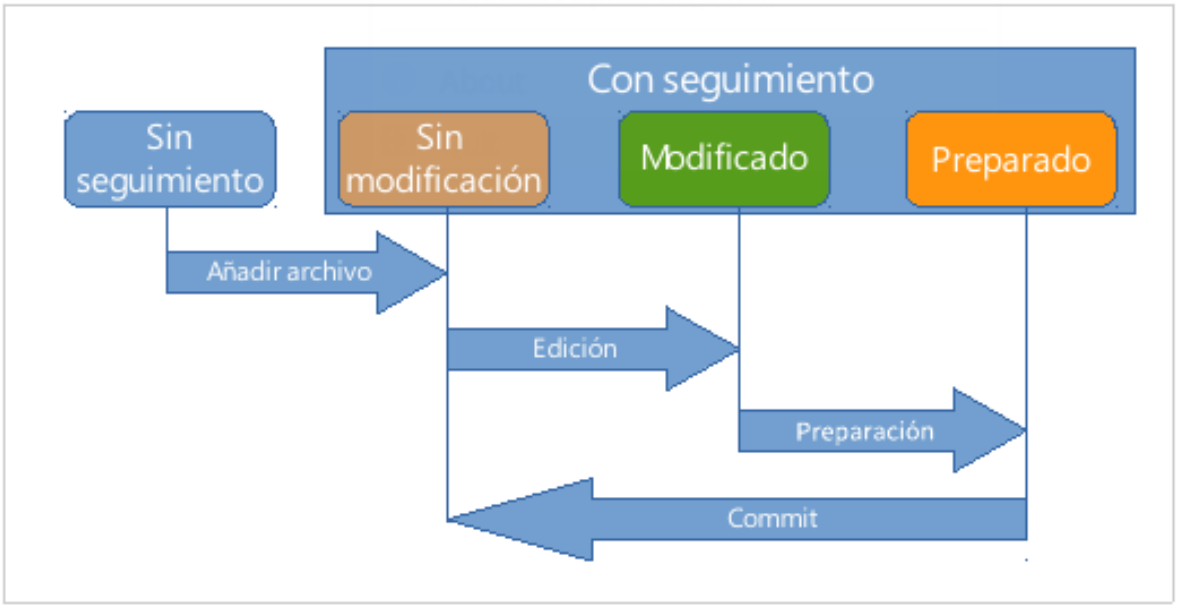

Figura 7: Estat d'un fitxer

- **SHA-1** i **HEAD**: Funció criptogràfica que aplicada a una seqüència de bytes de qualsevol tamany, com podria ser tot un fitxer o un conjunt de fitxers corresponents a un *commit*, dóna com a resultat un *ressum* d'aquest de 20 bytes (160 bits). Qualsevol modificació al fitxer, dóna un *ressum* completament diferent. Aquest és el mecanisme pel qual *git* sap quan modifiquem un fitxer, o bé quan un fitxer ha estat corromput, assegurant la *integritat* dels fitxers del repositori. Cada vegada que fem un *commit*, es calcularà un *SHA-1*, de manera que aquest identifique al propi *commit*. Cada *commit* que fem, llevat del primer, tindrà una referència al *commit* anterior. El *HEAD*, no és més que una referència que apunta a l'últim *commit* realitzat.
- **Branques** o *branches* són les bifurcacions, o camins alternatius que en un determinat moment agafem per resoldre algun bug, provar codi nou, o afegir noves funcionalitats, però sobre el que

no estem segurs que vaja a ser viable (és a dir, no estem segurs que el codi que anem a posar vaja a funcionar i potser haguem de desfer tot el que hem fet). En la majoria d'ocasions, treballarem a la branca principal o ***master*****, a partir de la qual podem crear altres branques i anar canviant entre elles, de manera que canvie el directori de treball**.

Flux de treball

El flux de treball, una vegada tinguem inicialitzat un repositori serà:

1. Creació de fitxers en el directori o subdirectoris del projecte. Aquest fitxer estarà *sense seguiment*.
2. Afegim els fitxers al control de versions. En aquest moment, el fitxer estarà *baix seguiment*, però *modificat*, ja que el seu contingut, en aquest directori de treball és diferent al de la zona de preparació i al repositori.
3. Editem el fitxer, afegint més canvis, el fitxer segueix estant *modificat*.
4. Preparem el fitxer, afegint una instantània d'aquest a la zona de preparació. Aquest estarà ara *preparat* per ser incorporat al repositori al següent *commit*. Ara el contingut de la zona de treball i de preparació coincideix, però difereix del del repositori.
5. Fem el *commit*, de manera que passem el fitxer al repositori, i torna a l'estat de *sense modificació* ja que coincideix en totes les zones de git.
6. Tornem al punt 3, on anem afegint noves modificacions.

Amb açò, cada *commit* que fem, anirà generant noves signatures SHA, i fent referència a l'anterior. El *HEAD*, sempre apuntarà a l'últim *commit* vàlid.

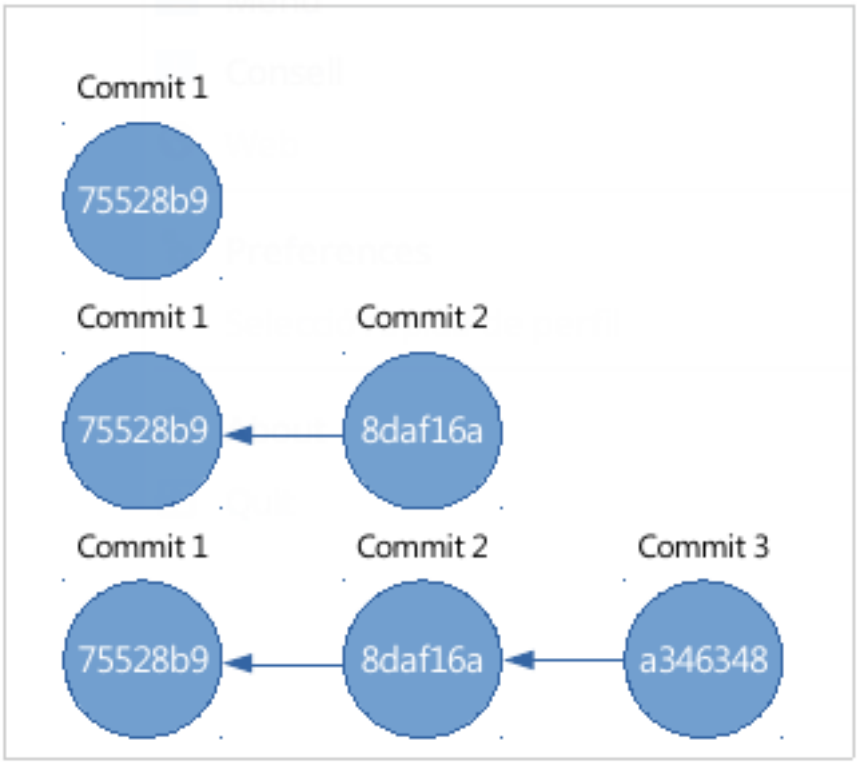

Figura 8: commits

Veiem un altre exemple de flux de treball amb branques:

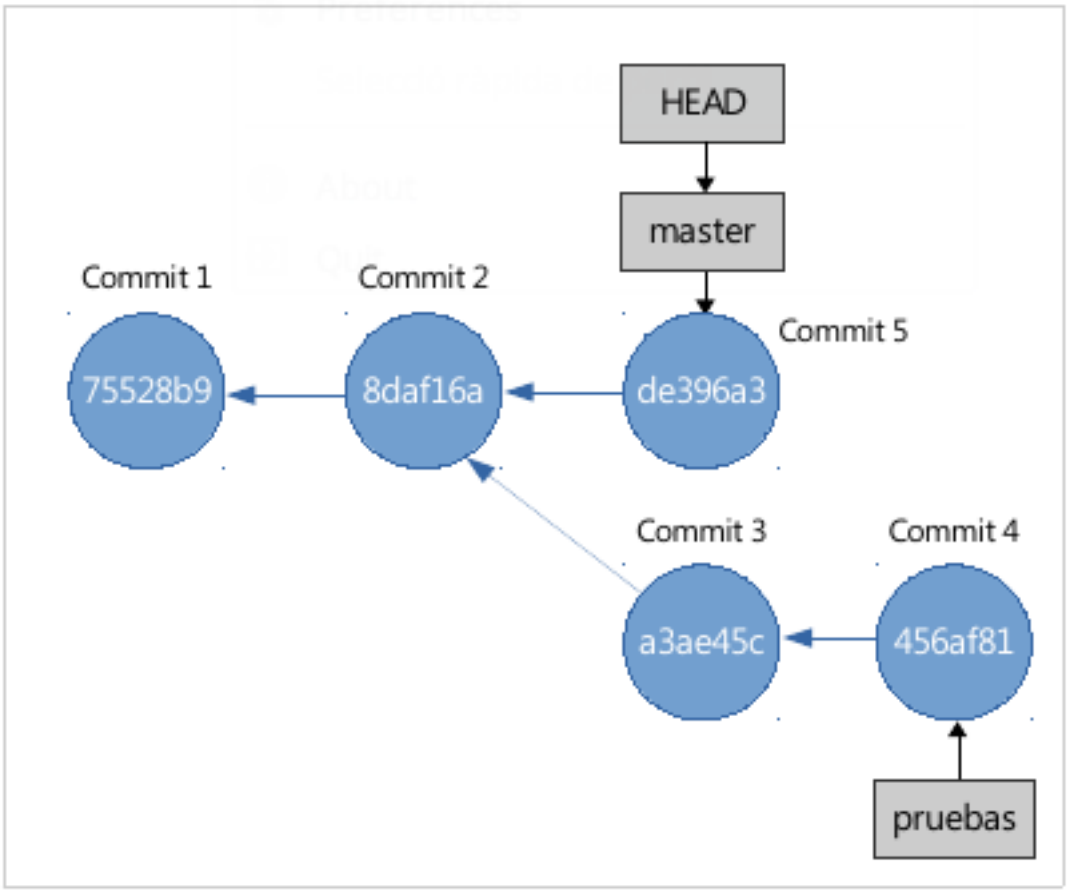

Figura 9: Branques

En aquest exemple, hi ha dos commit a la branca principal (*master*). Després, a partir del segon commit, s'ha creat una nova branca, anomenada *pruebas*, a la qual s'han afegit dos commits nous. Posteriorment, hem tornat a la branca *master* i hem afegit un nou *commit*. Com poeu vore, el *HEAD* apuntarà sempre al *commit* més recent.

3.5. Exemple pràctic

Anem a veure un xicotet exemple d'ús de git per il·lustrar tot el què hem comentat.

3.5.1. Creació i inicialització del projecte i el repositori

Per treballar amb un repositori git, el primer que hem de fer és inicialitzar-lo. Això ho podem clonant un repositori existent, o creant-ne un de nou. En aquest exemple, en crearem un de nou.

- Ens situem en el directori on anem a guardar els projectes, i creem la carpeta del projece amb:

```
$ mkdir projecte
```

Ara ens ubiquem dins d'ell:

```
cd projecte
```

I inicialitzem aci el repositori:

```
git init
```

Git ens dirà que ha inicialitzat un repositori buït.

Si volem veure l'estat de git, farem:

```
git status
```

I veurem el següent missatge:

```
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Com veiem, ens diu que estem a la branca *master*, que no hem fet cap *commit*, ni hi ha cap pendent.

Si feu un `ls -la`, per mostrar els fitxers ocults, veureu que apareix una carpeta `.git`, que conté tota la informació sobre el control de versions.

3.5.2. Creació d'un fitxer nou:

```
$ touch fitxer_1.md
```

Amb açò hem creat un fitxer buit. Si veiem l'estat de git, ens mostrarà:

```
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    fitxer_1.md

nothing added to commit but untracked files present (use "git add" to
    track)
```

Com veiem ens marca `fitxer_1.md` com a *Untracked*, el que vol dir que aquest fitxer es troba en estat de *sense seguiment*.

3.5.3. Fer el seguiment del fitxer

Per tal de marcar el fitxer per a que git en faça el seguiment, farem:

```
$ git add fitxer_1.md
```

Si ara consultem l'estat:

```
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   fitxer_1.md
```

Com veiem, ens indica que fitxer_1.md és un fitxer nou a git, i que té canvis per fer un *commit*. Açò ens mostra que el fitxer ha passat de l'estat *sense seguiment* a l'estat *Preparat*, el que vol dir que està a punt per poder fer un commit.

3.5.4. Fent el commit del fitxer

Per afegir, ara sí, el fitxer al repositori, fem el nostre primer commit:

```
git commit -a -m "Primer Commit"
[master (root-commit) 46c5f91] primer commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fitxer_1.md
```

Amb el `-a` indiquem que s'afisquen tots els canvis que hem fet, i amb el `-m` indiquem el missatge d'aquest commit. Si no espeficiquem missatge, ens obrirà l'editor per defecte per a que l'indiquem.

Fixeu-se en la primera línia que ens torna, on ens indica el SHA-1 d'aquest commit: 46c5f91.

Si ara fem un `git status`, veurem que ens diu que no hi ha res per actualitzar, i que el directori de treball està net, és a dir, que el que tenim al directori de treball es correspon amb el que hi ha al repositori.

```
$ git status
On branch master
nothing to commit, working tree clean
```

També podem veure un registre dels commits amb l'ordre:

```
$ git log
```

O bé de forma més simplificada en una línia amb:

```
$ git log --oneline
```

Exercici 1: Proveu a modificar el fitxer i observeu els canvis en l'estat. En quin estat es trobarà ara el fitxer? Pugeu els canvis de nou (només caldrà fer el commit, ja que el fitxer ja està baix el control de versions), i expliqueu el resultat. **Exercici 2**: Creeu nous fitxers, i comproveu l'status de git. Afegiu-los al repositori i pugeu els canvis, indicant rere cada ordre que feu en quin estat estan els fitxers.

3.5.5. Esborrant fitxers

L'esborrat de fitxers és una tasca bastant habitual quan treballem en un projecte. Amb *git*, podem eliminar fitxers de dues formes.

En primer lloc, anem a crear, dins el repositori dos fitxers per esborrar posteriorment:

```
$ touch tmp1.md
$ touch tmp2.md
```

Si veiem l'estat:

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    tmp1.md
    tmp2.md
```

Com veieu, ens indica que no s'està seguint estos dos fitxers. Per afegir-los al seguiment, farem:

```
$ git add .
```

I observem de nou amb `git status`que s'ha afegit i estan marcats com a nous:

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

    new file:   tmp1.md
    new file:   tmp2.md
```

Ara fem el commit:

```
$ git commit -a -m "Afegits dos fitxers de prova"
[master fb3bcef] Afegits dos fitxers de prova
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 tmp1.md
 create mode 100644 tmp2.md
```

Ara ja tenim els dos fitxers al repositori, pel que podem començar a veure com fer l'esborrat.

Per tal d'esborrar un fitxer de git, ho podem fer de dues formes:

- Esborrar-lo en local i després en *git*:
  - Esborrem el fitxer en local

```
$ rm tmp1.md
```

  - Veiem com l'status canvia a *deleted*

```
$ git status
On branch master
Changes not staged for commit:
(use "git add/rm <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working
    directory)

  deleted:    tmp1.md

no changes added to commit (use "git add" and/or "git commit -a")
```

  - Com veieu, ens diu que indiquem els canvis del pròxim commit amb `git add/rm fitxer`. Per tant:

```
$ git rm tmp1.md
```

  - Si are fem un `git status` ens indicarà que es marcarà aquest fitxer com eliminat per al pròxim commit:

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

  deleted:    tmp1.md
```

  - Ara ja podem fer el commit amb el fitxer esborrat:

```
$ git commit -a -m "Esborrat tmp1.md"
[master 5e3b2e1] Esborrat tmp1.md
```

```
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 tmp1.md
```

- Esborrar-lo directament en git i provocar el seu esborrat en la zona de treball i preparació, deixant-lo llest per fer el commit.
  - Esborrem directament l'ordre d'esborrat a git:

```
$ git rm tmp2.md
```

  - Si are observem l'status, veurem com només hem de fer el commit per tal que es reflexen els canvis. Fixeu-se també que el fitxer ha desaparegut de l'àrea de treball:

```
$ git commit -a -m "Esborrat tmp2.md"
```

3.5.6. Movent fitxers

Per tal de moure (o renomenar) fitxers, també podem fer-ho de les dues formes anteriors: primer al sistema i passant-ho a git, o directament a git. Anem a veure directament com fer-ho des de git:

- En primer lloc, creem un fitxer, l'afegim al control de versions i el pugem al repositori:

```
$ touch mv1.md
$ git add mv1.md
$ git commit -a -m "Afegit mv1.md"
[master b063c63] Afegit mv1.md
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename tmp2.md => mv1.md (100%)
```

Ara, per moure el fitxer, farem ús de `git mv`:

```
$ git mv mv1.md mv1_renomenat.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    mv1.md -> mv1_renomenat.md
```

Com veiem, comprovant l'status ens diu que el fitxer ha estat renomenat, i que aquest renomenament es farà efectiu al pròxim commit. Només ens queda doncs, pujar-lo:

```
$ git commit -a -m "Renomenat mv1.md a mv1_renomenat.md"
[master 3ec28c1] Renomenat mv1.md a mv1_renomenat.md
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename mv1.md => mv1_renomenat.md (100%)
```

3.5.7. Desfent canvis entre la zona de preparació i treball

Quan tenim canvis pendents a la zona de preparació, podem desfer-los utilitzant de nou un *checkout*.

Per exemple, si modifiquem l'anterior fitxer `mv1_renomenat.md` afegint noves línies:

```
$ echo "Povant desfer canvis" >> mv1_renomenat.md
```

I consultem l'estat:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
     directory)

    modified:   mv1_renomenat.md
```

Veiem que el marca com modificat. Si ara el que volem és desfer aquests canvis que ja tenim en la zona de preparació, farem un `git checkout --` del fitxer:

```
$ git checkout -- mv1_renomenat.md
```

I si ara consultem l'estat, veurem que ja tornem a tindre-ho tot com abans de la modificació de la línia:

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

3.6. Altres aspectes d'interès

3.6.1. Ignorant fitxers: El fitxer .gitignore

Moltes vegades no voldrem que determinats tipus de fitxers es guarden al repositori: Executables, fitxers comprimits resultats de l'empaquetat, directoris de llibreries, directoris o fitxers amb configuració, contrasenyes o paràmetres de connexió, etc.

Per a això, existeix el fitxer `.gitignore`, que guardem a la pròpia arrel del directori de treball, i en el que especificarem els diferents patrons de noms de fitxer que no s'inclouran mai al control de versions.

Aquest fitxer conté diverses línies que seguiran les següents regles:

- S'accepten línies en blanc com a separador per millorar la legibilitat.
- Els comentaris comencen amb #,
- El prefix ! nega el patró (farà que sí que s'incloguen a git els fitxers indicats)
- Podem utilitzar el comodí * per referenciar *qualsevol cadena de caràcters*
- Podem utilitzar la interrogació (?) per reemplaçar un caràcter qualsevol.
- Podem utilitzar expressions regulars tipus [0-9], [ao].

Veiem un exemple d'aquest tipus de fitxer `.gitignore`:

```
# Ignora els fitxers amb nom temporal_6.txt i temporal_7.zio
temporal_6.txt
temporal_7.zip

# Ignora tots els fitxers amb extensió zip, gz, changes o deb:
*.zip
*.gz
*.changes
*.deb

# Ignora tots els fitxers amb extensió .log de la carpeta log, així
# com les extensions .log0, .log1, log2...
log/*.log
log/*.log[0-9]

# Ignora tots els fitxers del directori imatges
imatges/*

# Ignora tots els fitxers que acaben en 'a' o 'o' de la carpeta
   compilats
compilats/*[ao]
```

Ara només ens queda afegir al repositori git aquest fitxer i pujar els canvis.

- **Exercici**: *Creeu un fitxer `.gitignore` i comproveu, mitjançant la creació de diversos fitxers que no deurien afegir-se als commits que aquest funciona correctament.*

3.6.2. Altres operacions

Veiem algunes altres operacions a destacar d'entre tot el què podem fer amb *git*.

- **Revertir a l'estat d'un *commit***:

```
git revert SHA_DEL_COMMIT_A_REVERTIR
```

Exemple:

- Veiem el llistat de commits:

```
$ git log --oneline
  3ec28c1 (HEAD -> master) Renomenat mv1.md a mv1_renomenat.md
  b063c63 Afegit mv1.md
  5e3b2e1 Esborrat tmp1.md
  fb3bcef Afegits dos fitxers de prova
  46c5f91 primer commit
```

- Revertim el commit *5e3b2e1*:

```
$ git revert 5e3b2e1
```

Que ens indicarà que especfiquem el missatge per al canvi i farà un nou commit amb el missatge indicat i els canvis aplicats.

- Veiem de nou el llistat de commits:

```
$ git log --oneline
5205a7a (HEAD -> master) Revert "Esborrat tmp1.md"
3ec28c1 Renomenat mv1.md a mv1_renomenat.md
b063c63 Afegit mv1.md
5e3b2e1 Esborrat tmp1.md
fb3bcef Afegits dos fitxers de prova
46c5f91 primer commit
```

- Per altra banda, si volem revertir només un fitxer d'un commit, ho podem fer amb `git checkout COMMIT -- fitxer_a_revertir`.
- **Eliminar fitxers no seguits:**

```
$ git clean -f
```

Amb aquesta ordre eliminem tots els fitxers del directori de treball que no estan sent seguits per git. Si a més volem que esborre també els directoris no seguits, farem:

```
$ git clean -f -d
```

3.7. Clonat de repositoris: Github

Anteriorment, hem comentat com crear un repositori a partir d'un directori. Anem a veure ara com obtenir un repositori remot (clonat), que és com treballarem a Github.

Quan parlem de *clonar* un repositori ens referim a portar al nostre equip un espai de treball i un repositori remot complet, de manera que rebem tot el conjut de commits del projecte i d'altres elements.

En general, per tal de clonar un projecte farem:

```
git clone URL [directori]
```

Així, per exemple, si volem clonar un repositori que es troba en una carpeta del nostre mateix equip, farem:

```
$ git clone exemple_inicial/projecte exemple_inicial/projecte2
```

Per altra banda, si el que volem és clonar un repositori remot, el que farem serà connectar-nos a ell, bé a través d'*HTTPS*, o bé de *SSH*. Per exemple, podem fer:

```
$ git clone ssh://usuari@127.0.0.1/home/jose/exemples_git
    exemple_inicial/projecte ./exemple_inicial/projecte3

Cloning into './exemple_inicial/projecte3'...
jose@127.0.0.1's password:
remote: Counting objects: 13, done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 13 (delta 3), reused 0 (delta 0)
Receiving objects: 100% (13/13), done.
Resolving deltas: 100% (3/3), done.
```

Aquesta és la manera que tindrem de *clonar* un repositori de github. Github és una plataforma de programari lliure que ens serveix de magatzem de projectes de codi lliure a través de git.

Activitat pràctica

Accediu al lloc web de github (www.github.com) i creeu-vos un compte gratuit.

Una vegada fet, creeu un repositori nou de zero (amb el desplegable amb el símbol + que apareixerà la part dreta de la barra superior). Aquest repositori tindrà un nom tipus *elTeuUsuari/EDD*. Quan el creeu, inclogueu un fitxer README.md, i observeu la interfície que us oferix.

Busqueu el botó *Clone or download*, i observeu que us apareixerà una adreça *https* a la qual podreu fer un `git clone`.

Feu aquest clonat a la vostra carpeta del mòdul on aneu a publicar totes les vostres memòries de pràctiques d'EDD.

Podeu trobar ajuda al següent vídeo, que us explica com crear un repositori en github (a partir del min. 37): https://www.youtube.com/watch?v=HiXLkL42tMU. En ell us conta una altra forma de sincronitzar-vos amb ell diferent a la que veurem.

Sincronitzant-nos amb el repositori remot

Ja sabem com crear repositoris, tant local com remots, i fer un clonat d'ells. Sabem també com afegir commits, i guardar-los al nostre repositori local. Només ens queda com *sincronitzar-nos* amb els repositoris remots de manera que el nostre repositori i el repositori remot tinguen el mateix contingut.

Per a això, farem ús de les següents ordres:

- Per **actualitzar el nostre repositori local des del repositori remot original** amb el que estiguem sincronitzats (ja hem fet un clonat): `git pull`
- Per **enviar els commits que hem fet en local al repositori remot original**, una vegada sincronitzats, utilitzarem: `git push`.

També ens pot ser d'utilitat l'ordre `git remote -v`, per veure els repositoris remots que tenim configurats, Per exemple:

```
$ git remote -v
origin  https://github.com/IesElJust/Bionic.git (fetch)
origin  https://github.com/IesElJust/Bionic.git (push)
```

Activitat pràctica 2

Una vegada clonat al vostre equip el repositori que heu creat, creeu la següent estructura de carpetes, a la qual haureu de penjar les pràctiques de les diferents unitats:

```
.
|-- imatges
|-- readme.md
|-- Unitat_1
|-- Unitat_2
`-- Unitat_3
```

Com veieu, crearem una carpeta per a les imatges de totes les pràctiques. Si ho veieu més pràctic per organitzar-vos, podeu crear una carpeta d'imatges per a cada unitat. Una vegada tingueu pujades les imatges, aquestes tindran una URL (no la pàgina de github que les conté, sinò la imatge en sí). Aquesta URL és la que haureu d'utilitzar per incloure-les al vostre codi en markdown.

Recordeu que per incloure una imatge al Markdown ho feu amb:

```
![Descripció de la imatge](URL_DE_LA_IMATGE)
```