UNIDAD 3.
SISTEMAS DE CONTROL DE VERSIONES (VCS). GIT

VICENT MARTÍ

OBJETIVOS

- Conocer los VCS más utilizados.
- Comprender la utilidad de las características de los VCS.
- Conocer como utilizar los VCS. GIT.

CONTENIDO

1. Introducción a los VCS
2. Sistemas de control de versiones OpenSource
3. Características de los VCS
4. Modelos de control de versiones
5. Terminología
6. Formas de colaborar
7. Ejemplos de comandos
8. GIT

1. INTRODUCCIÓN A LOS VCS

Cuando hablamos de control de versiones en programación nos referimos a la **capacidad de registrar los cambios realizados sobre los archivos del código fuente**. Este control puede hacerse de manera manual creando diferentes archivos para cada modificación o realizando copias incrementales con la fecha de modificación pero esto puede llevar mucho tiempo y dar como resultado un sistema desorganizado por lo que es mejor automatizar el proceso.

Los **VCS** (version control system) son aplicaciones que permiten realizar el control de modificaciones de código fuente de **manera automática** y eficiente. 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.

Estos enlaces a Wikipedia te muestran las principales características y podrás acceder a las páginas oficiales.

2. SISTEMAS DE CONTROL DE VERSIONES OPENSOURCE

GIT: Funciona con un modelo distribuido y está licenciado bajo GNU GPL. Originalmentre fue diseñado por Linus Torvalds y es utilizado para el proyecto del núcleo de Linux.

Subversion: Funciona con un modelo centralizado y está licenciado bajo Apache.

CVS: Funciona con un modelo centralizado y está licenciado bajo GNU GPL. Este proyecto tiene mantenimiento pero no se agregarán nuevas características.

Mercurial: Funciona con un modelo distribuido y está licenciado bajo GNU GPL.

CVS - Concurrent Versions System

3. CARACTERÍSTICAS DE LOS VCS

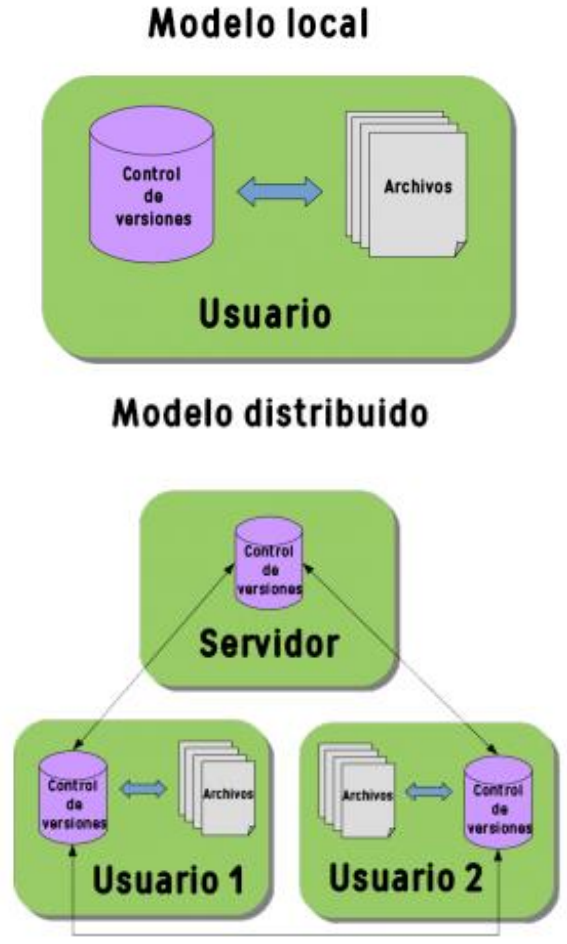
El control de versiones se realiza principalmente en la industria informática para controlar las distintas versiones del **código fuente** dando lugar a los **sistemas de control de código fuente** o SCM (siglas del inglés *Source Code Management*). Sin embargo, los mismos conceptos son aplicables a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

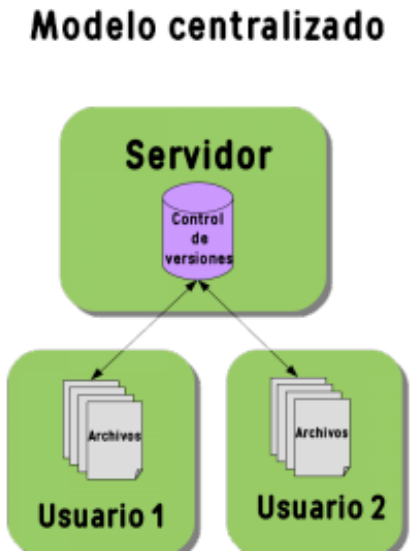
Algunas de las características de los VCS son:

- **Reporte de cambios:** Cuando un archivo es modificado se guarda la fecha y hora del cambio, el autor y un mensaje opcional explicando las razones o naturaleza del cambio.
- **Sincronización:** Varias personas en el mismo proyecto que todas puedan tener la última versión
- **Backup y restauración:** Permite guardar los cambios realizados en los archivos y restaurar el archivo a cualquiera de los estados previos en los que fue guardado.
- **Crear branch:** Permite crear una branch (rama) del proyecto y trabajarla por separado sin modificar el proyecto original.
- **Realizar merge:** Permite convertir el contenido de branch al proyecto principal
- **Generación de informes**: Aunque no es estrictamente necesario, con los cambios introducidos entre dos versiones, informes de estado, marcado con versión de un conjunto de ficheros, etc.

4. MODELOS DE CONTROL DE VERSIONES

- **Modelo local:** utiliza una copia de la base de control de versiones y una copia de los archivos del proyecto. Este tipo es el más sencillo y no es recomendable cuando se trabaja en equipo ya que todos tienen que acceder a los mismos archivos.
- **Modelo centralizado:** el control de versiones se realiza en un servidor que se encargará de recibir y dar los cambios realizados en el archivo a cada uno de los usuarios.
- **Modelo distribuido:** es el más utilizado, en este caso cada usuario tiene un control de versiones propio que a su vez son manejadas por el servidor.

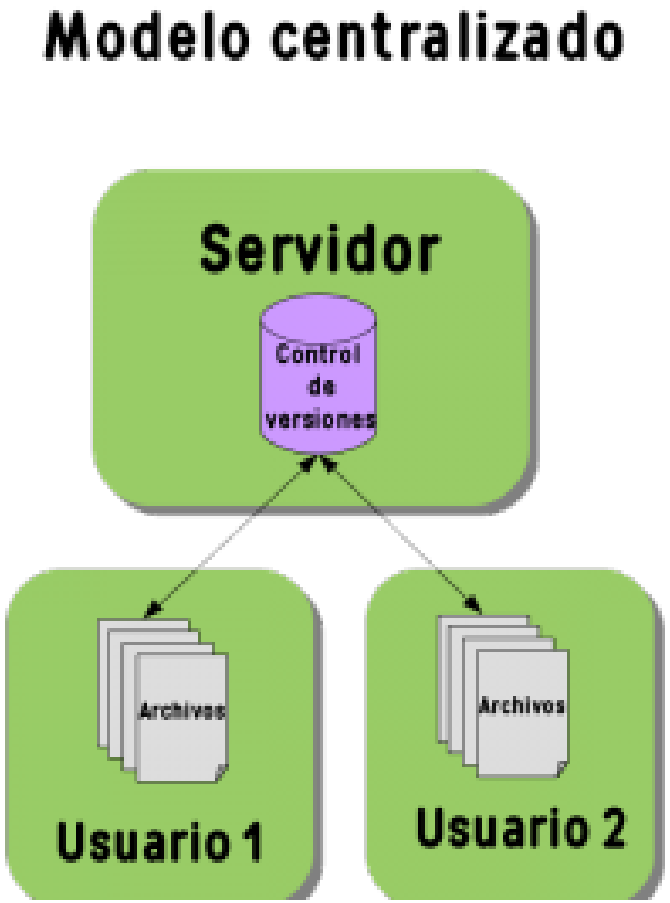




MODELO CENTRALIZADO I

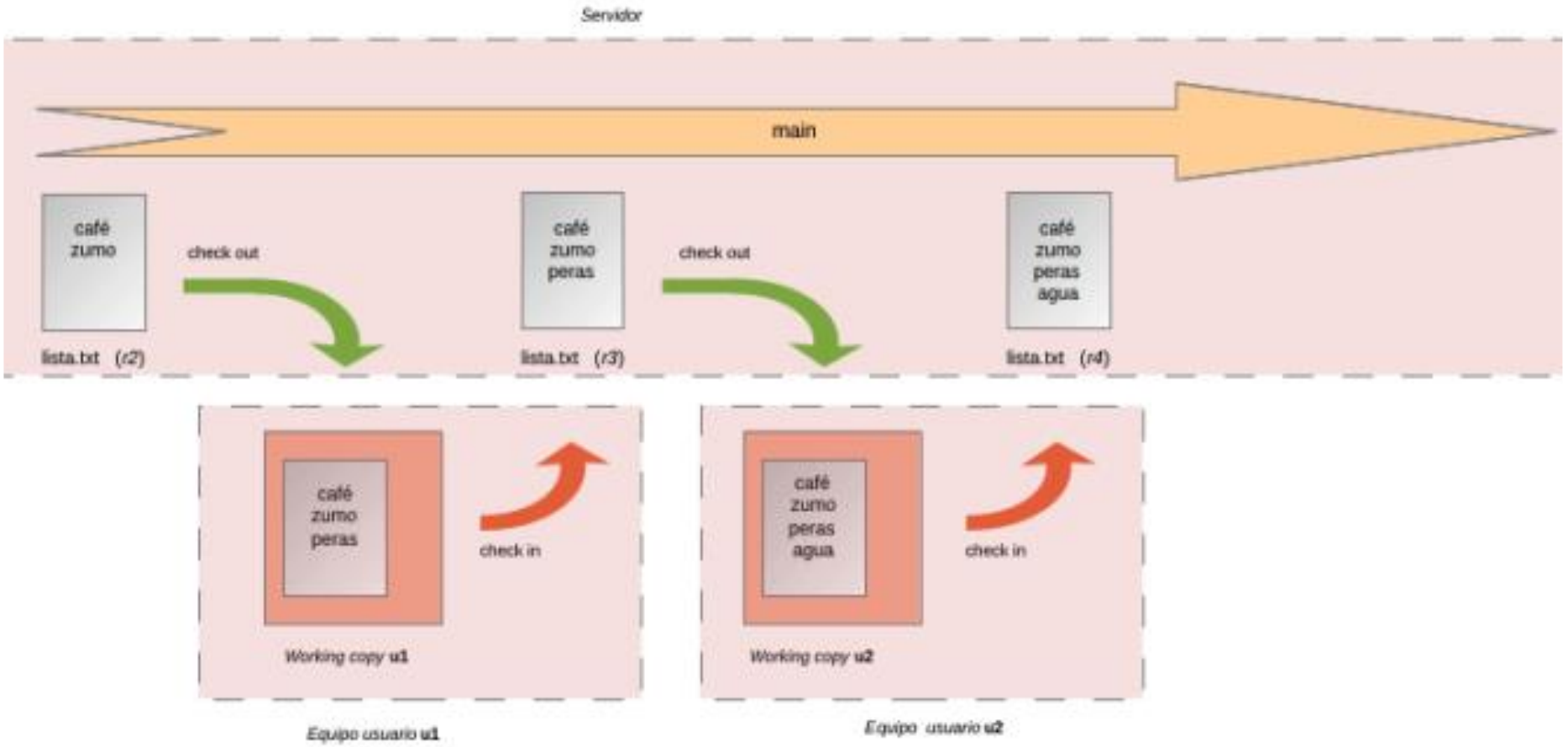
Usa un servidor central para almacenar el repositorio así como el control de acceso al mismo. El repositorio se mantiene separado de la copia que existe en el directorio de trabajo de cada usuario. Suele ser un servidor dedicado (o al menos un ordenador distinto al de cualquier usuario), aunque en proyectos pequeños o con un único usuario se suele asignar otra carpeta del ordenador de trabajo.

Con un sistema centralizado, la copia de trabajo sólo almacena la versión actual de los ficheros del proyecto y las modificaciones que el usuario esté realizando en ese momento. La única copia de la historia completa está en el repositorio. Cuando un usuario envía (checkin) cambios al repositorio central, en ese momento ya podrán ser descargados por el resto de usuario



MODELO CENTRALIZADO II

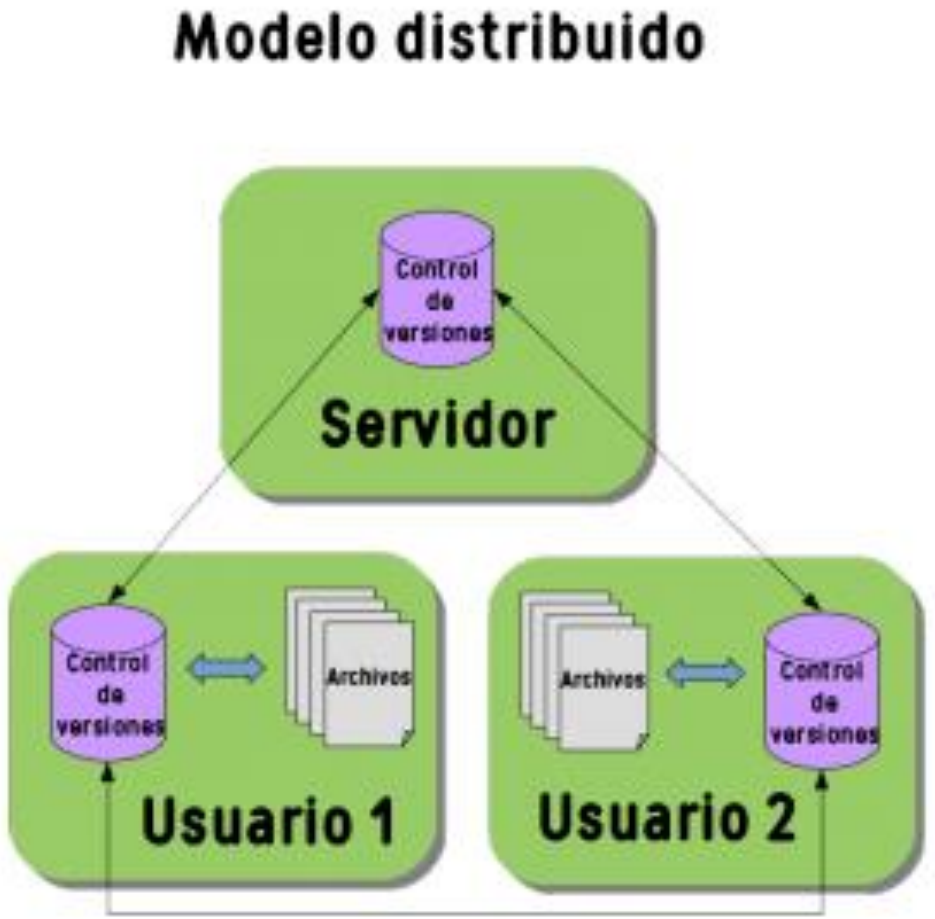
Como puede verse en el esquema, todos los usuarios hacen su checkin y checkouts sobre la rama principal: u1 añade *peras* y u2 añade *agua*. Para que los cambios de u1 puedan ser visto por u2, u1 tiene que hacer checkin al servidor.



MODELO DISTRIBUIDO I

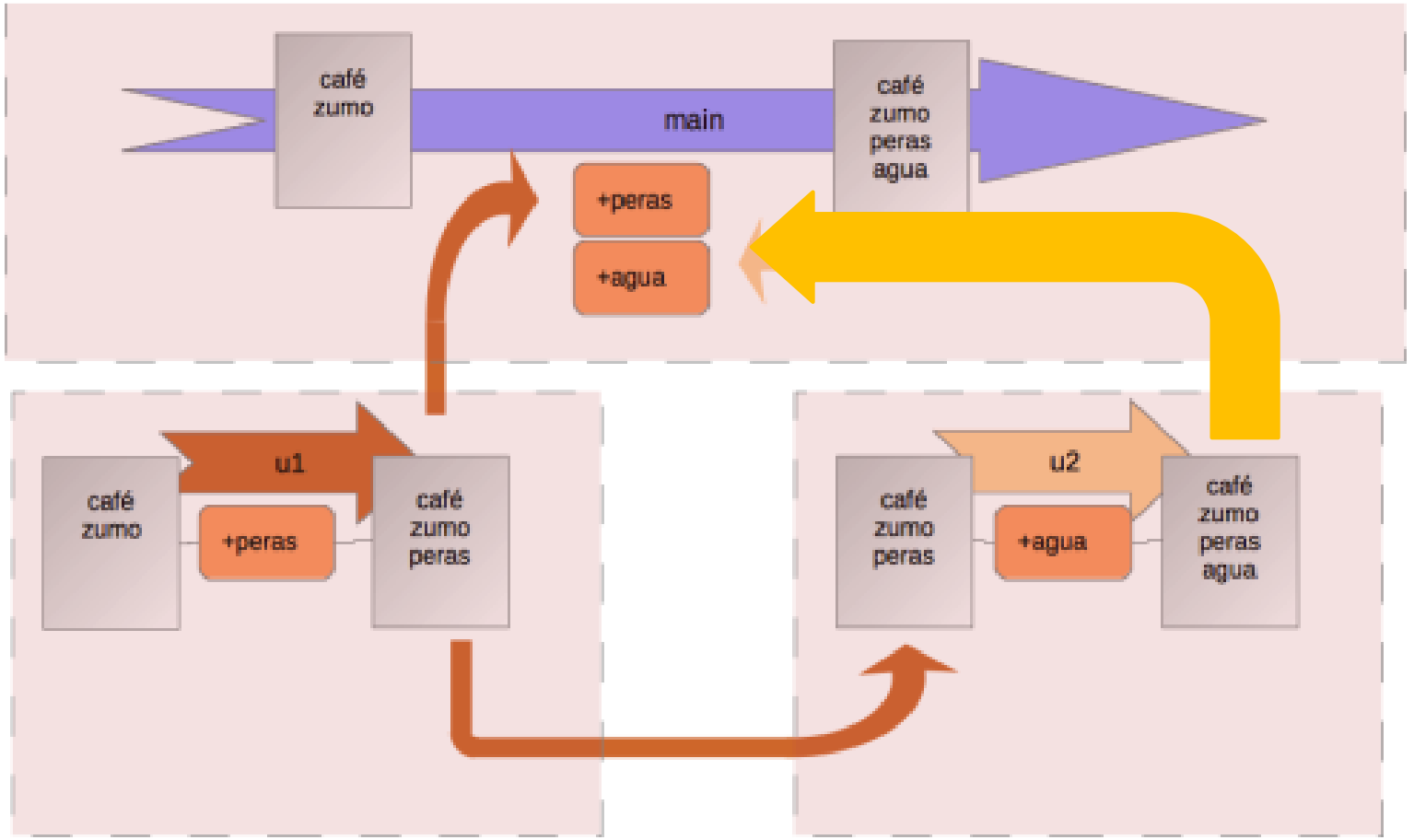
En un sistema distribuido cada usuario tiene su propio repositorio. Los cambios que hace u1 estarán en un repositorio local que podrá compartir o no con el resto de usuarios. Todos los usuarios son iguales entre si, el concepto maestro/esclavo que aparece en los centralizados desaparece.

Aparentemente esta forma de trabajar da la sensación de ser un poco caótica y es por ello que suele ser habitual que además exista un repositorio central donde se sincronizan todos los cambios locales de cada usuario. Cada uno de los usuarios realiza los checkout y chekin con respecto a su propio repositorio.



MODELO DISTRIBUIDO II

Cada uno de los usuarios realiza los checkout y chekin con respecto a su propio repositorio. A esos cambios sólo tendrá acceso cada usuario (cada uno al suyo respectivamente) hasta que decida compartirlos con uno o con el resto de usuarios (o con el servidor). Esto es una diferencia con respecto al sistema centralizado en el que al hacer un chekin todos los usuarios tiene acceso a la modificación



VENTAJAS DE SISTEMAS DISTRIBUIDOS

- Necesitan estar conectados menos veces a la red para hacer operaciones, esto produce una mayor autonomía y 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. 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.
- 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.

INCONVENIENTES DE SISTEMAS DISTRIBUIDOS

- No hay claramente una última versión del código. En un sistema distribuido en el que no haya un repositorio central no queda claro de qué se compone la versión última versión estable.
- No existen números de revisión definidos. En un sistema distribuido en realidad no hay números de versión, ya que cada repositorio tiene sus propios números dependiendo de la cantidad de cambios que se han realizado en él. Por contra, como se ha comentado antes, existe un identificador de la forma e4e561f3 por cada cambio, mas complicado de recordar.

5. TERMINOLOGÍA I

La terminología empleada puede variar de sistema a sistema, pero a continuación se describen algunos términos de uso común.

Repositorio: es el lugar en el que se almacenan los datos actualizados e históricos de cambios, a menudo en un servidor. A veces se le denomina depósito o depot. Pueden ser archivos en un disco duro, un banco de datos, etc.

Módulo: Conjunto de directorios y/o archivos dentro del repositorio que pertenecen a un proyecto común.

Revisión (*"version"*): es una versión determinada de la información que se gestiona. Hay sistemas que identifican las revisiones con un contador (Ej. subversion). Hay otros mediante un código de detección de modificaciones (Ej. Git usa SHA1). A la última versión se le suele identificar con el nombre de **HEAD**. Para marcar una revisión concreta se usan los **rótulos** o *tags*.

5. TERMINOLOGÍA II

Rotular (*"tag"*): Darle a alguna versión de cada uno de los ficheros del módulo en desarrollo en un momento preciso un nombre común ("etiqueta" o "rótulo") para asegurarse de reencontrar ese estado de desarrollo posteriormente bajo ese nombre. Los tags permiten identificar de forma fácil revisiones importantes en el proyecto. Por ejemplo se suelen usar tags para identificar el contenido de las versiones publicadas del proyecto. En algunos sistemas se considera un tag como una rama en la que los ficheros no evolucionan, están congelados.

Línea base (*"baseline"*): Una revisión aprobada de un documento o fichero fuente, a partir del cual se pueden realizar cambios subsiguientes.

Tronco o principal: línea principal de código en el repositorio.

Abrir rama (*"branch"*) o ramificar o **bifurcado:** en un instante de tiempo de forma que, desde ese momento en adelante se tienen dos copias (ramas) que evolucionan de forma independiente siguiendo su propia línea de desarrollo. El módulo tiene entonces 2 (o más) "ramas". La ventaja es que se puede hacer un "merge" de las modificaciones de ambas ramas, posibilitando la creación de "ramas de prueba" que contengan código para evaluación, si se decide que las modificaciones realizadas en la "rama de prueba" sean preservadas, se hace un "merge" con la rama principal. Son motivos habituales para la creación de ramas la creación de nuevas funcionalidades o la corrección de errores.

5. TERMINOLOGÍA II

Desplegar (*"Check-out"*, *"checkout"*, "co"): Un despliegue crea una copia de trabajo local desde el repositorio. Se puede especificar una revisión concreta, y predeterminadamente se suele obtener la última.
Conflicto: Un conflicto ocurre cuando el sistema no puede manejar adecuadamente cambios realizados por dos o más usuarios en un mismo archivo.
Resolver: El acto de la intervención del usuario para atender un conflicto entre diferentes cambios al mismo archivo.
Cambio (*"change"*, *"diff"*, *"delta"*): Un **cambio** representa una modificación específica a un archivo bajo control de versiones. La granularidad de la modificación considerada un cambio varía entre diferentes sistemas de control de versiones.
Lista de cambios (*"changelist"*, *"change set"*, *"patch"*): En muchos sistemas de control de versiones con *commits* multi-cambio atómicos, una lista de cambios identifica el conjunto de cambios hechos en un único *commit*. Esto también puede representar una vista secuencial del código fuente, permitiendo que el fuente sea examinado a partir de cualquier identificador de lista de cambios particular.

5. TERMINOLOGÍA III

Exportación (*"export"*): Una exportación es similar a un ***check-out***, salvo porque crea un árbol de directorios limpio sin los metadatos de control de versiones presentes en la copia de trabajo. Se utiliza a menudo de forma previa a la publicación de los contenidos.

Importación (*"import"*): Una importación es la acción de copia un árbol de directorios local (que no es en ese momento una copia de trabajo) en el repositorio por primera vez.

Congelar: Permitir los últimos cambios (*commits*) para solucionar las fallas a resolver en una entrega (*release*) y suspender cualquier otro cambio antes de una entrega, con el fin de obtener una versión consistente. Si no se congela el repositorio.

5. TERMINOLOGÍA IV

Integración o fusión (*"merge"*): Una **integración** o **fusión** une dos conjuntos de cambios sobre un fichero o un conjunto de ficheros en una revisión unificada de dicho fichero o ficheros. Puede suceder:

- Cuando un usuario, trabajando en esos ficheros, **actualiza** su copia local con los cambios realizados, y añadidos al repositorio, por otros usuarios. Análogamente, este mismo proceso puede ocurrir en el repositorio cuando un usuario intenta ***check-in*** sus cambios.
- Después de que el código haya sido ***branched***, y un problema anterior al *branching* sea arreglado en una rama, y se necesite incorporar dicho arreglo en la otra.
- Después de que los ficheros hayan sido ***branched***, desarrollados de forma independiente por un tiempo, y que entonces se haya requerido que fueran fundidos de nuevo en un único *trunk* unificado.

Integración inversa: El proceso de fundir ramas de diferentes equipos en el *trunk* principal.

Actualización (*"sync"* o *"update"*): Una **actualización** integra los cambios que han sido hechos en el repositorio (por ejemplo por otras personas) en la **copia de trabajo** local.

Copia de trabajo (*"workspace"*): es la copia local de los ficheros de un repositorio, en un momento del tiempo o revisión específicos. Todo el trabajo realizado sobre los ficheros en un repositorio se realiza sobre una copia de trabajo, de ahí su nombre. Conceptualmente, es un **cajón de arena** o ***sandbox***.

5. TERMINOLOGÍA V

La propia idiosincrasia de un sistema distribuido hace que se identifiquen determinadas acciones como:

- Empujar (push): envía un cambio desde un repositorio a otro. Según cual sea la política del sistema podría ser necesario tener los permisos adecuados.
- Extraer (pull): coge los cambios desde un repositorio.
- Clonar (clone): traer una copia exacta del proyecto desde un repositorio a otro.

6. FORMAS DE COLABORAR

Para colaborar en un proyecto usando un sistema de control de versiones lo primero que hay que hacer es crearse una **copia local** obteniendo información del repositorio. A continuación el usuario puede modificar la copia. Existen dos esquemas de funcionamiento:

- **De forma exclusiva:** para poder realizar un cambio es necesario comunicar al repositorio el elemento que se desea modificar y el sistema se encargará de impedir que otro usuario pueda modificar dicho elemento. Una vez hecha la modificación, esta se comparte con el resto de colaboradores. Este modo de funcionamiento es el que usa por ejemplo SourceSafe. Otros sistemas de control de versiones como Subversion, aunque no obligan a usarlo, disponen de mecanismos que permiten implementarlo.
- De forma **colaborativa**: cada usuario modifica la copia local y cuando el usuario decide compartir los cambios el sistema automáticamente intenta combinar las diversas modificaciones. El principal problema es la posible aparición de conflictos que deban ser solucionados manualmente o las posibles inconsistencias que surjan al modificar el mismo fichero por varias personas no coordinadas. Además, esta semántica no es apropiada para ficheros binarios. Los sistemas Subversion o Git permiten implementar este modo de funcionamiento.

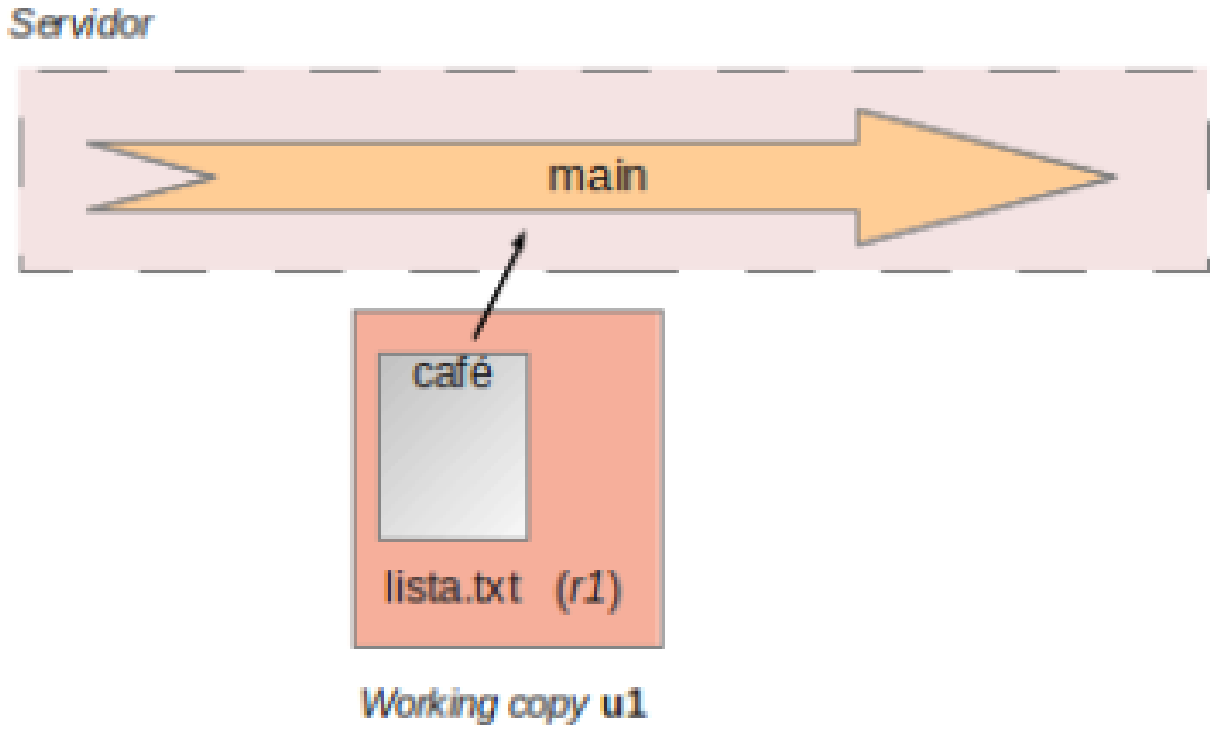
7. EJEMPLOS DE COMANDOS

Podemos ver como algunos de los comandos más utilizados actúan en un Sistema de Control de Versiones.

- Add: añadir lista.txt al control de versiones
- Check out (for edit), check in (commit) y revert.
- Tags: etiquetar cada versión.
- Diff: obtener las diferencias.
- Branch: abrir ramas.
- Merge: mezclar ramas.
- Conflictos: falta información para decidir.

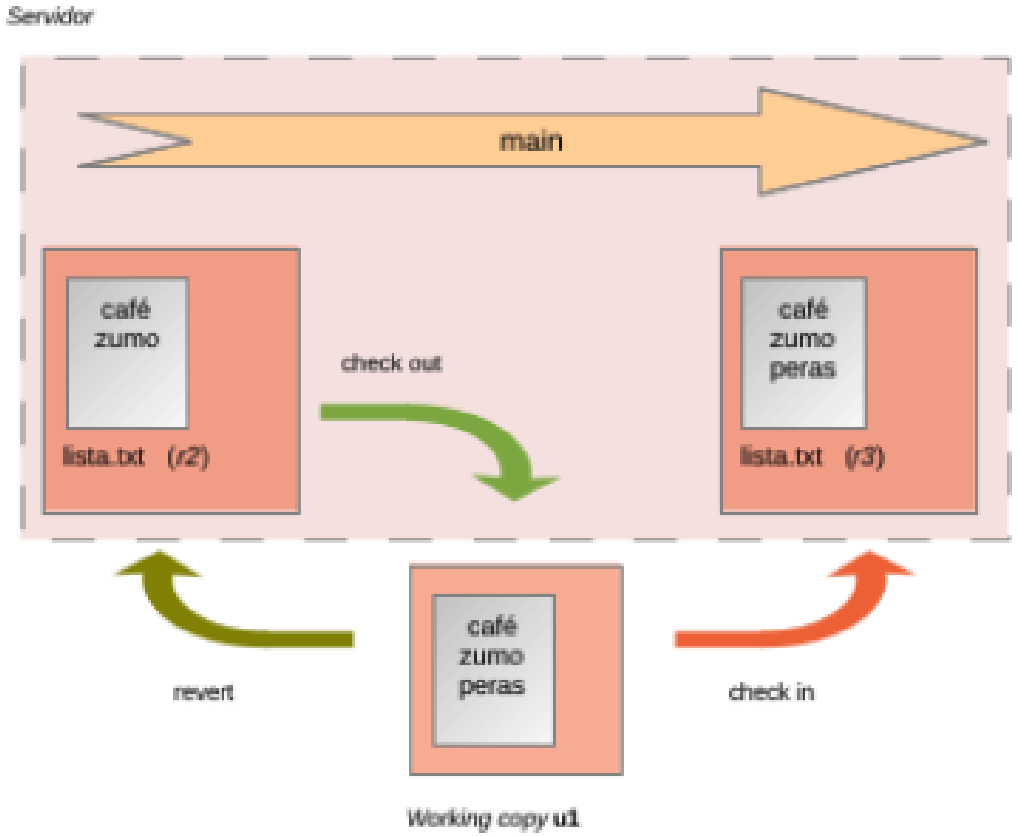
ADD

Lo primero que hay que hacer es añadir lista.txt al control de versiones mediante una comando de Add. En este caso podemos ver como el usuario u1, desde su directorio de trabajo incluye el fichero lista.txt a la línea principal (main) del servidor donde se encuentra el sistema de control. El fichero lista.txt ya tiene un ítem (café), pero podría estar vacío. Automáticamente el VCS le asigna un numero de revisión (r1).



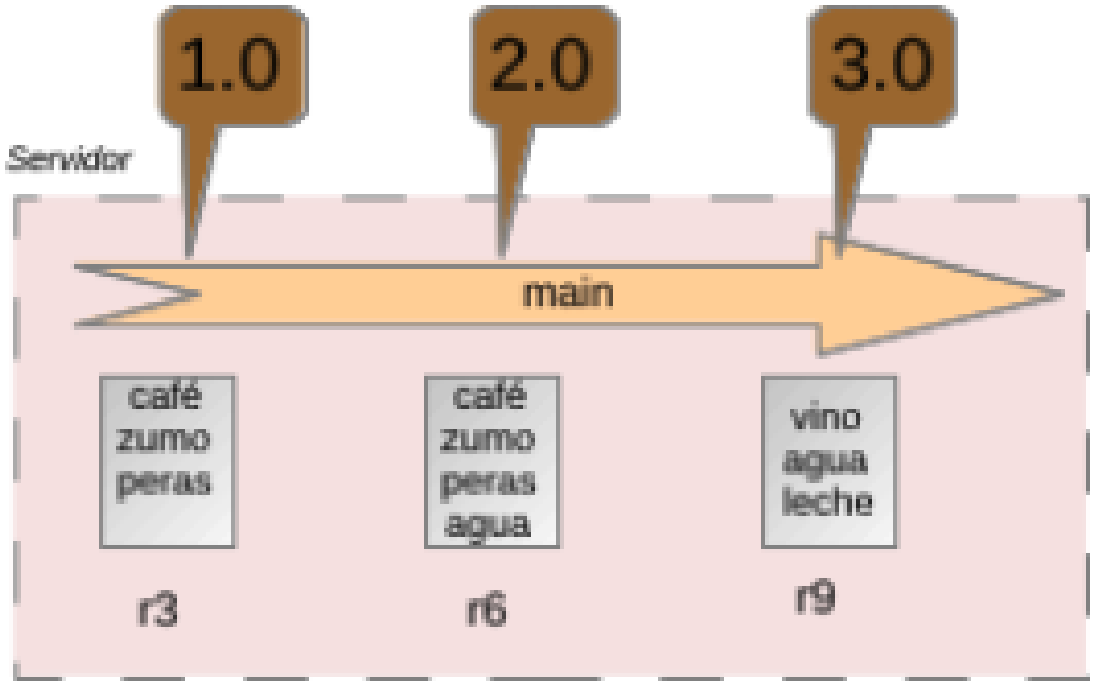
CHECK OUT (FOR EDIT), CHECK IN (COMMIT) Y REVERT

u1 va modificando el fichero a lo largo de la semana, incluyendo nuevos ítems a la lista. Para ello, **u1** realiza primero un check out for edit. Esta operación en realidad son dos operaciones: en primer lugar se actualiza el fichero lista.txt del directorio de trabajo a la última versión existente en el servidor (r1) y posteriormente se activa la posibilidad de editarlo. Una vez u1 tiene a su disposición la copia, la edita y añade un ítem (peras) y, una vez añadido, lo actualiza en el servidor mediante un chek in (o commit) creando una nueva revisión (r3). Es posible que **u1** una vez modificado el fichero, se de cuenta de un error y no quiera perder los cambios y volver a la última versión del servidor para lo cual realizaría una acción revert.



TAGS

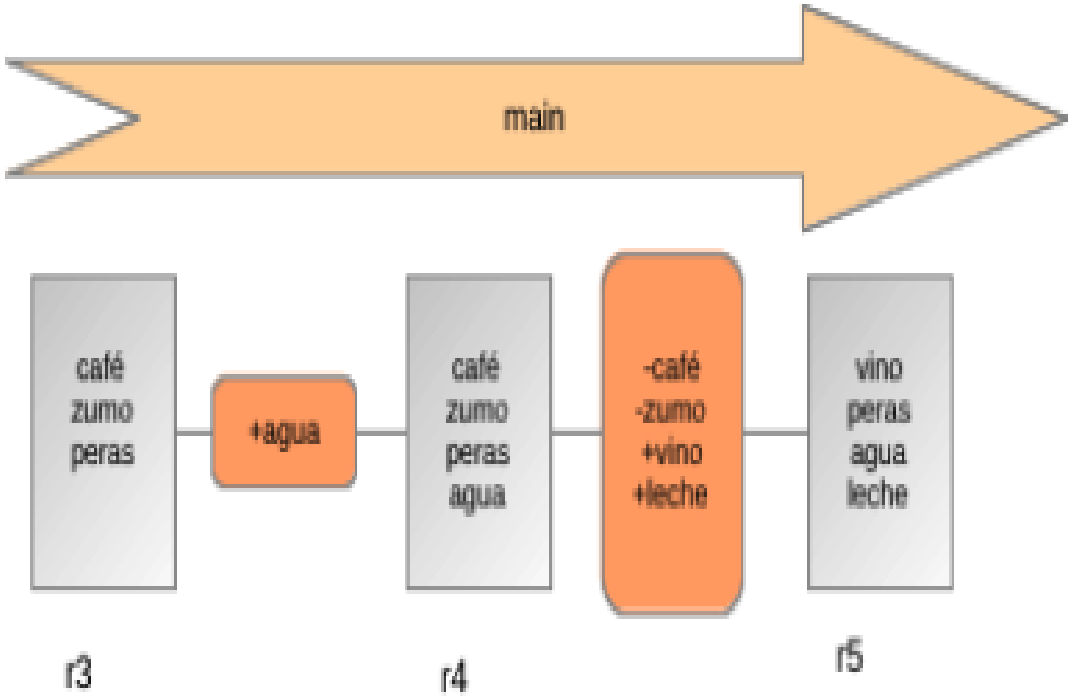
Puede ser interesante que a final de cada semana, antes de empezar con la lista de la semana siguiente, etiquetar cada versión del fichero para saber que fue lo que se compro en una determinada fecha.



DIFF

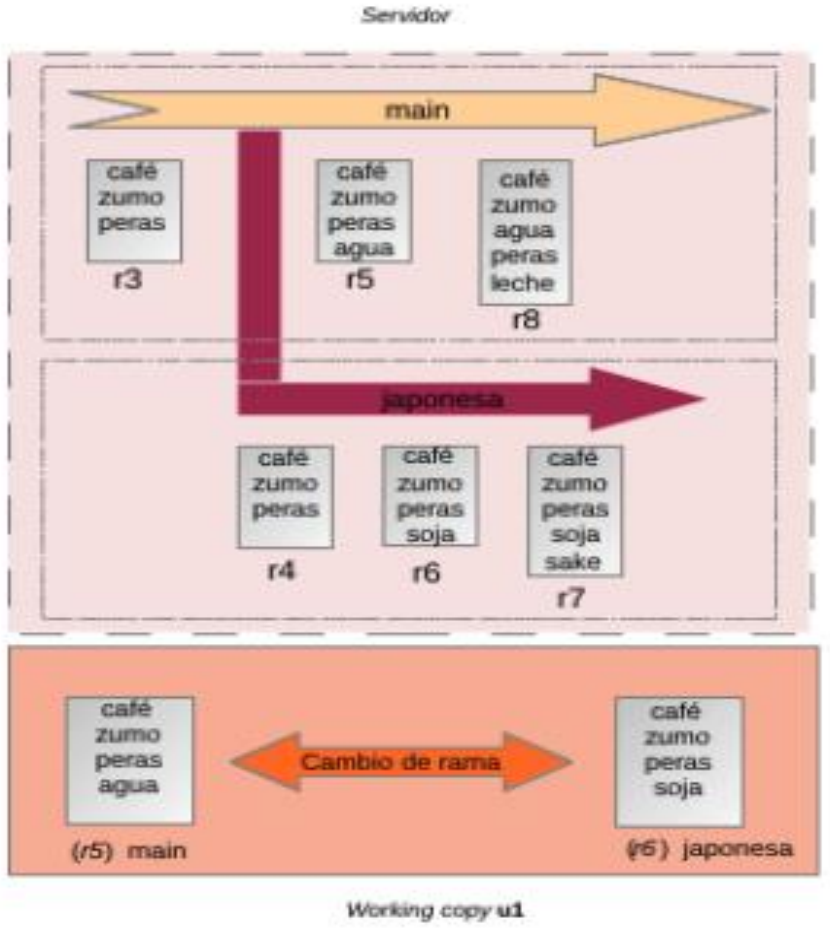
Uno de los usuarios tiene interés en ver en qué se ha modificado la lista entre dos versiones. Para obtener las diferencias, el VCS se hace preguntas como ¿qué debo
modificar en una versión para llegar a otra?. Si nos fijamos en el ejemplo, para llegar a r5 a partir de r4, habría que añadir (+) vino y leche y eliminar (-) café y zumo.

Las consultas pueden hacerse entre cualesquiera versiones si bien cuanto mas alejadas en el tiempo estén, por lo general, mas tiempo tarda el sistema en calcular las diferencias y mas complejo será seguirlas. Además, el VCS da las diferencias por líneas o bloques de líneas consecutivas, es decir, indica que en la línea 1 en la versión r4 había café y en la línea 1 de r5 se ha quitado (-) café y se ha añadido (+) vino.



BRANCH

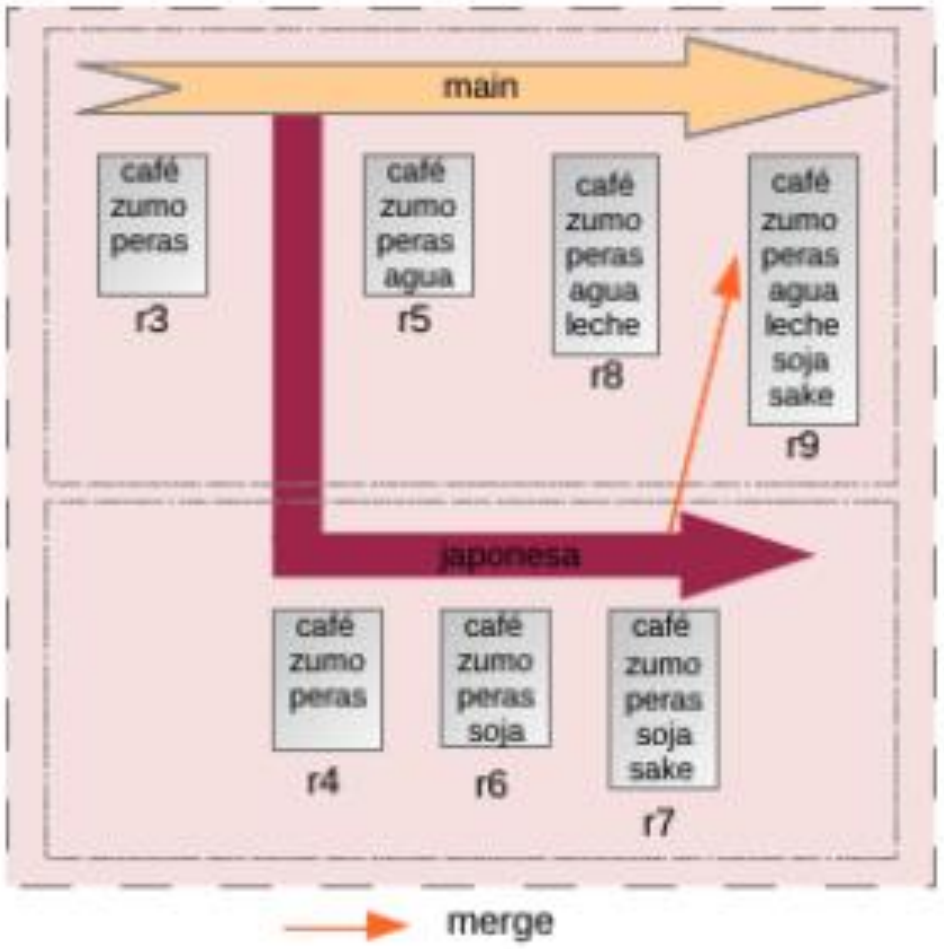
u1 decide que quiere experimentar cocinando comida japonesa. Para ello abre una rama llamada japonesa, donde ira incluyendo ítems relacionados con ese tipo de comida, hasta que una semana decida ponerse a ello. Al abrir una rama lista.txt puede evolucionar por dos caminos distintos *main* y *japonesa*. El punto de partida de japonesa (r4) es una copia de la revisión existente en main en el momento de abrir la rama. A partir de ese momento u1 puede elegir sobre qué rama trabajar. Si desea incluir algo en la rama japonesa, **u1** tiene que ir a esa rama, hacer un *chekout*, modificar y hacer un *chekin* sobre la rama. Posteriormente para seguir añadiendo cosas en la lista del día a día debe volver a la rama main.



MERGE

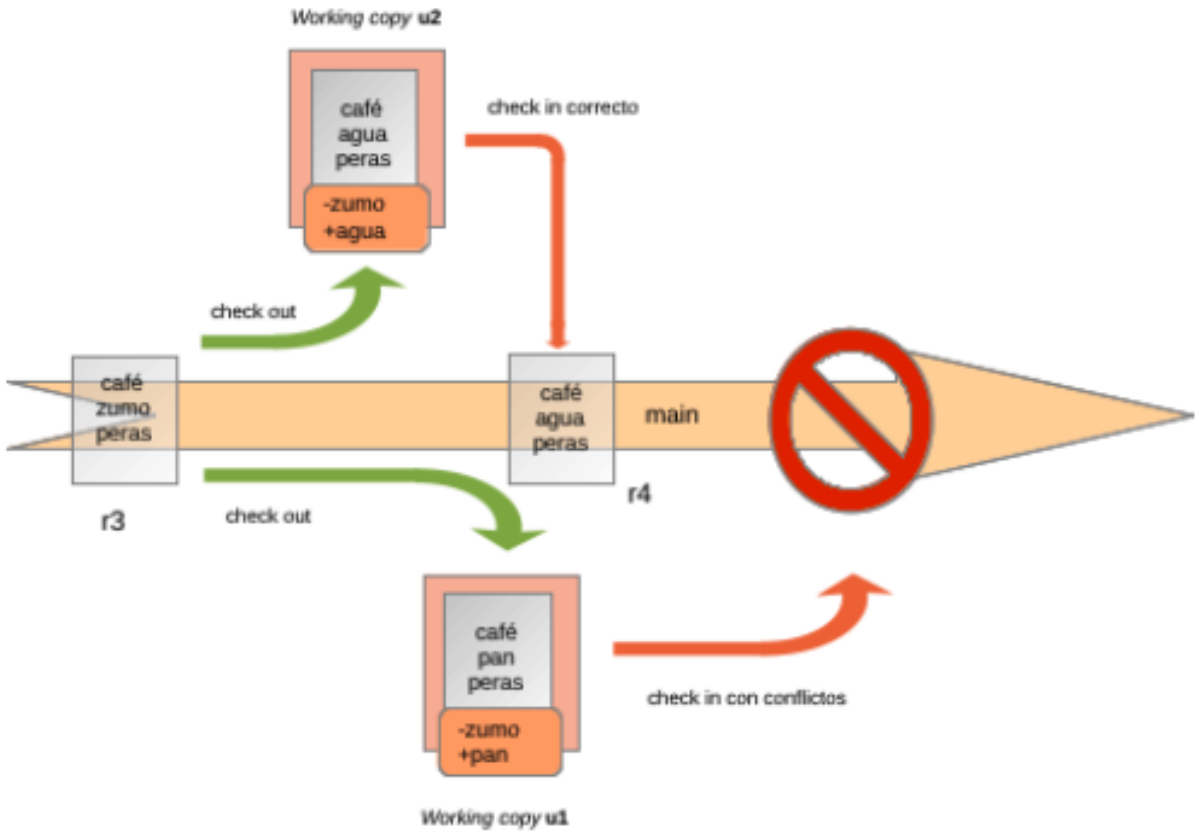
u1 por fin se decide y la semana que viene esta dispuesto a cocinar comida japonesa, por lo que ha de mezclar (merge) la lista de la rama japonesa con la rama principal que es de donde se obtiene la lista con la que se ira al mercado.

Cuando el usuario mezcla una rama con otra, el VCS calcula las diferencias entre el principio de la rama a mezclar (o desde la última versión del fichero que fue mezclada) y la versión final y aplica dichas diferencias a la versión actual de la rama destino. En el ejemplo se calcula la diferencia entre r7y r4.es decir añadir (+) soja y sake. El mezclado no sólo implica la existencia de dos ramas. En el caso de que dos usuarios estén editando el mismo fichero, cuando ambos realizan el *checkin*, el VCS realiza automáticamente el merge.



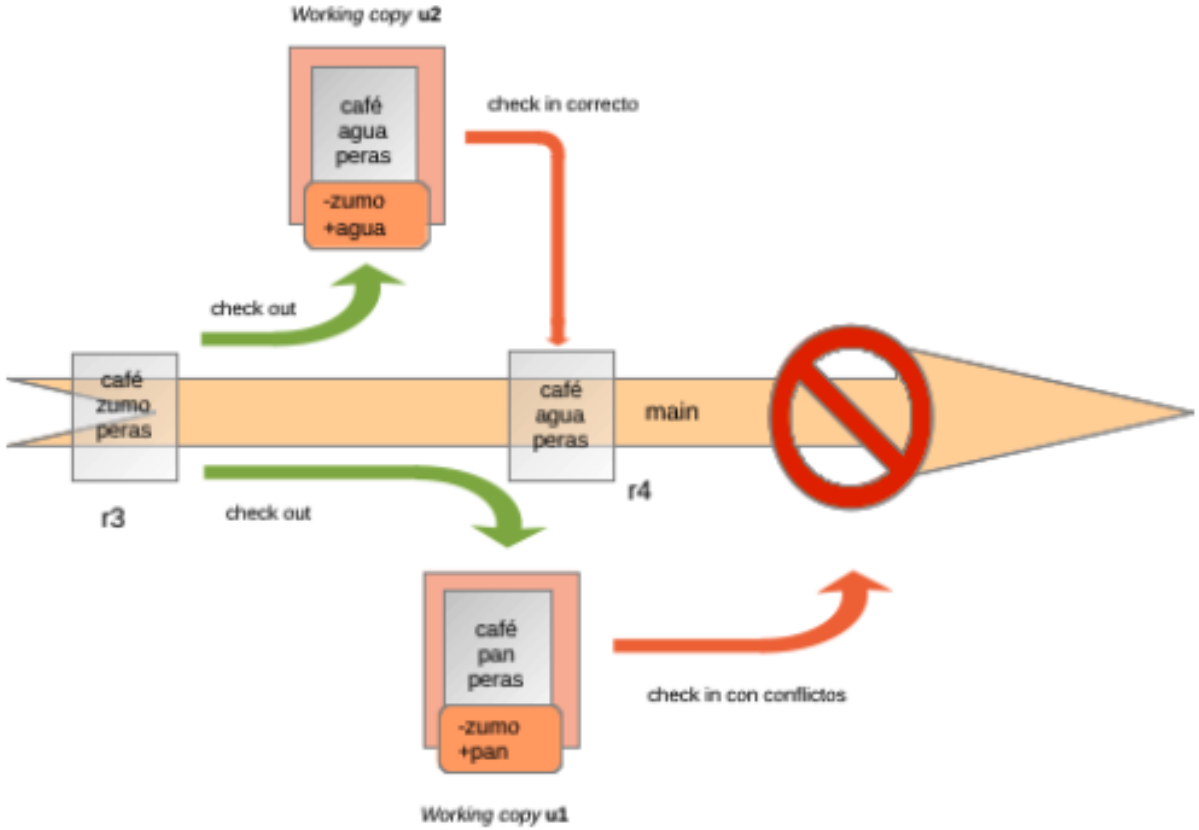
CONFLICTOS I

En la gran mayoría de casos el propio VCS mezcla las versiones sin mayor problema, pero puede darse casos en que esto no sea así. Tanto **u1** como **u2** deciden que quieren cambiar la lista y para ello hacen un check out de manera mas o menos simultanea. u2 no quiere zumo y en cambio quiere agua, mientras que u1 que tampoco quiere zumo, quiere pan en su lugar. u2 cambia realiza el cambio y el checkin antes que u1. Cuando u1 va a realizar el chekin, su copia local tiene que mezclarse con la r4 pero el sistema detecta que desde que se hizo el checkout (r3) otro usuario ha cambiado las mismas líneas (en este caso la n°2). El VCS no tiene información suficiente para decidir con cual quedarse (¿agua o pan?) y genera un conflicto.



CONFLICTOS II

En este caso solicita al usuario **u2** que resuelva el conflicto, es decir, que manualmente decida cual es la línea que deberá permanecer o que incluya las dos líneas. En condiciones normales la aparición de conflictos no es (o al menos no debería ser) muy habitual. Con un poco de organización, distribución ordenada del trabajo y haciendo checkin de manera mas o menos habitual, su aparición disminuye mucho. Aun así, tal como se comentó anteriormente, si se va a editar gran parte del fichero y durante mucho tiempo, puede ser recomendable que el usuario lo bloqueé para evitar la edición por parte de otros.



EJERCICIO 1

Escribe un documento y súbelo al aula virtual (AV)

- Escoge 4 CVS (no escojas GIT) y analiza las diferentes características.

8. GIT

Git (pronunciado "guit") es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente. Su propósito es llevar registro de los cambios en archivos de computadora y coordinar el trabajo que varias personas realizan sobre archivos compartidos.

Al principio, Git se pensó como un motor de bajo nivel sobre el cual otros pudieran escribir la interfaz de usuario o front end, sin embargo, Git se ha convertido desde entonces en un sistema de control de versiones con funcionalidad plena. Hay algunos proyectos de mucha relevancia que ya usan Git, en particular, el grupo de programación del núcleo Linux.

Git es un software libre distribuible bajo los términos de la versión 2 de la Licencia Pública General de GNU.

Linus Benedict Torvalds es un ingeniero de software finlandés-estadounidense, conocido por iniciar y mantener el desarrollo del kernel Linux.

GIT: SISTEMA DE CONTROL DE VERSIONES I

Es una aplicación que permite a diferentes programadores trabajar en un proyecto común ocupándose cada uno de una parte y con un control centralizado. Es ideal para grandes proyectos o también para trabajar con otras personas que se puedan encontrar lejos físicamente.

Podemos decir que Git, hace una “copia” del proyecto, cuando confirmas un cambio, o guardas el estado del proyecto en Git, básicamente hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea. Tú decides cuando deseas hacer una foto a tu código, y esto se realiza mediante un “commit”.

Ventajas:

- La fácil distribución del código.
- Facilidad para coordinar el trabajo entre mÚltiples desarrolladores.
- Podemos saber quién y cuando se han hecho cambios.
- Revertir a versiones anteriores
- Git está disponible para los tres grandes sistemas operativos, Linux/Unix, Windows y Mac OS X.

GIT: SISTEMA DE CONTROL DE VERSIONES II

Los tres estados de GIT

Cuando trabajamos con GIT, nuestra información puede estar en una de las siguientes situaciones:

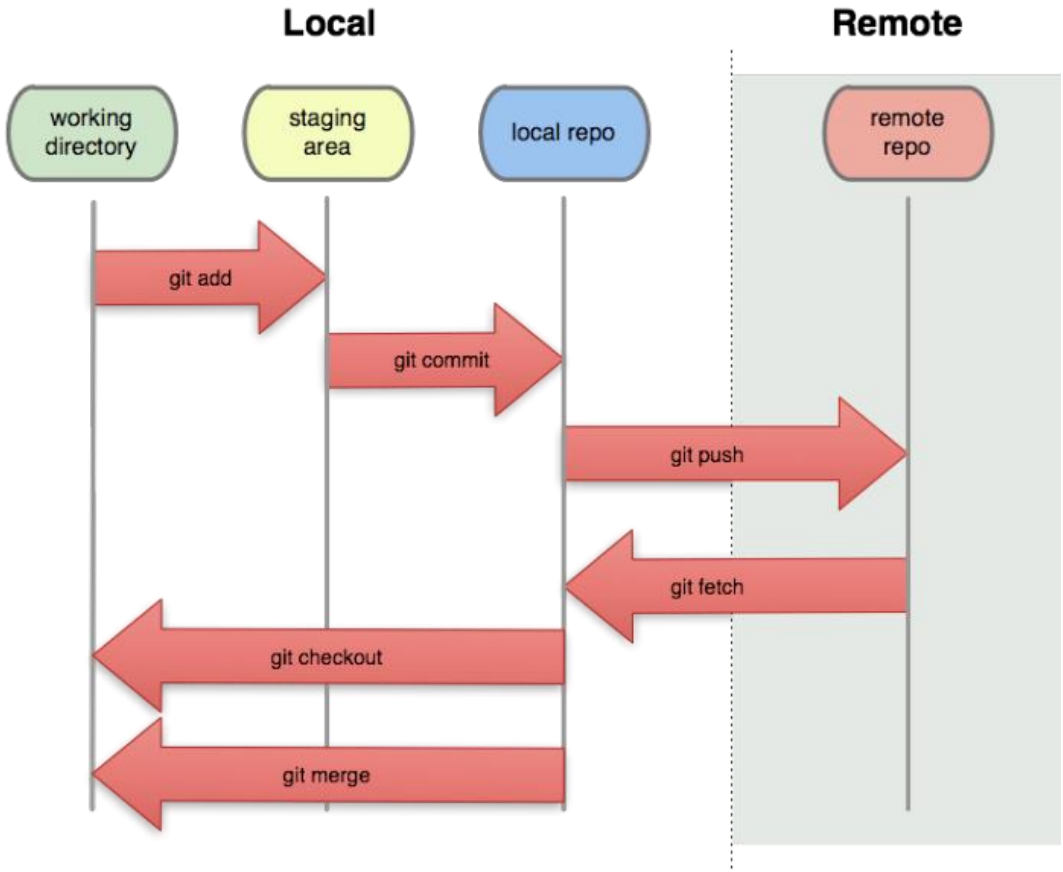
- Información modificada (**modified**). Este estado implica que hemos modificado nuestra información, pero aún no está siendo trackeada por GIT.
- Información preparada (**staged**). Este estado implica que hemos marcado nuestra información para posteriormente ser confirmada y por tanto trackeada como nueva versión.
- Información confirmada (**committed**). Este estado implica que nuestra información ha sido almacenada en la base de datos local de GIT.

Estas situaciones dan lugar a lo que se conoce como los tres estados de GIT, el **working copy**, el **stagin area** y el **repositorio**.

GIT: SISTEMA DE CONTROL DE VERSIONES III

- **Working copy**, área de trabajo en la que hacemos los cambios en nuestros ficheros.
- **Staging area**, espacio donde colocaremos aquellos ficheros listos para ser colocados en el repositorio.
- **Repositorio local**, área donde GIT irá guardando las distintas versiones de nuestra información.
- **Repositorio remoto**, área en la nube (github) para guardar versiones de nuestra información.

Todo nuestro trabajo con GIT se reduce a ir moviendo la información de un área a otra, gestionando de esa forma sus distintos estados. Para ello usaremos preferiblemente la línea de comandos de nuestro terminal aunque también existen utilidades gráficas que nos facilitan nuestro trabajo diario con GIT.



CARACTERÍSTICAS MÁS RELEVANTES

Entre las se encuentran:

- Fuerte apoyo al **desarrollo no lineal**, rapidez en la gestión de ramas y mezclado de diferentes versiones: Incluye herramientas específicas para navegar y visualizar un historial de desarrollo no lineal.
- Gestión **distribuida**: Git le da a cada programador una copia local del historial del desarrollo entero, y los cambios se propagan entre los repositorios locales. Los cambios se importan como ramas adicionales y pueden ser fusionados en la misma manera que se hace con la rama local.
- Los almacenes de información pueden publicarse por **HTTP, FTP, rsync** o protocolo nativo, ya sea a través de una conexión **TCP/IP o cifrado SSH**. Git también puede emular servidores CVS.
- Los repositorios Subversion y svk se pueden usar directamente con **git-svn**.
- Gestión eficiente de **proyectos grandes**.
- Todas las versiones previas a un cambio determinado, implican la notificación de un cambio posterior en cualquiera de ellas (denominado **autenticación criptográfica de historial**).
- Los **renombrados** se trabajan basándose en similitudes entre ficheros, y **evita posibles coincidencias de ficheros** diferentes en un único nombre.
- El **realmacenamiento periódico** en paquetes (ficheros).

GITHUB

GitHub es una plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git. Se utiliza principalmente para la creación de código fuente de programas de ordenador. El software que opera GitHub fue escrito en Ruby on Rails. Desde enero de 2010, GitHub opera bajo el nombre de *GitHub, Inc.* Anteriormente era conocida como *Logical Awesome LLC*. El código de los proyectos alojados en GitHub se almacena típicamente de forma pública.

El 4 de junio de 2018 Microsoft compró GitHub por la cantidad de 7500 millones de dólares. Es la plataforma más importante de colaboración para proyectos Open Source.

Ruby on Rails, es un framework de aplicaciones web de código abierto escrito en el lenguaje de programación Ruby, siguiendo el paradigma del patrón Modelo Vista Controlador (MVC). Trata de combinar la simplicidad con la posibilidad de desarrollar aplicaciones del mundo real escribiendo menos código que con otros frameworks y con un mínimo de configuración.

FLUJO DE TRABAJO

Git plantea una gran libertad en la forma de trabajar en torno a un proyecto. Sin embargo, para coordinar el trabajo de un grupo de personas en torno a un proyecto es necesario acordar como se va a trabajar con Git. A estos acuerdos se les llama **flujo de trabajo**. Un flujo de trabajo de Git es una fórmula o una recomendación acerca del uso de Git para realizar trabajo de forma uniforme y productiva. Los flujos de trabajo más populares son:

- git-Flow
- GitHub-Flow
- GitLab Flow
- One Flow

GIT-FLOW

Creado en 2010 por Vincent Driessen. Es el flujo de trabajo más conocido. Está pensado para aquellos proyectos que tienen entregables y ciclos de desarrollo bien definidos.

Basado en dos grandes ramas con infinito tiempo de vida (**ramas master y develop**) y varias ramas de apoyo:

- **Feature**: desarrollo de nuevas funcionalidades.
- **Hotfix**: arreglo de errores
- **Release**: preparación de nuevas versiones de producción.

GITHUB-FLOW

Creado en 2011 por GitHub y se ajusta a sus funcionalidades. Está centrado en un modelo de desarrollo iterativo y de despliegue constante.

Principios básicos:

- Todo lo que está en la **rama master está listo para ser puesto en producción**.
- Para trabajar en **algo nuevo, debes crear una nueva rama** a partir de la rama master con un nombre descriptivo. El trabajo se irá integrando sobre esa rama en local y regularmente también a esa rama en el servidor.
- Requerir información o integración de una rama local en la rama master, se debe abrir una **pull request (solicitud de integración de cambios).**
- Revisión de los **cambios para fusionar con la rama master**.

GITLAB FLOW

Creado en 2014 por Gitlab. Es una especie de extensión de GitHub Flow, pero trata de estandarizar aún más el proceso. Al igual que GitHub Flow propone el uso de ramas de funcionalidad (feature) que se originan a partir de la rama master y que al finalizarse se mezclan con la rama master.
Además introduce otros tipos de ramas:

- **Ramas de entorno**. Por ejemplo pre-production production. Se crean a partir de la rama master cuando estamos listos para implementar nuestra aplicación. Si hay un error crítico lo podemos arreglar en un rama y luego mezclarla en la rama de entorno.
- **Ramas de versión.** Por ejemplo 1.5-stable 1.6-stable. El flujo puede incluir ramas de versión en caso de que el software requiera lanzamientos frecuentes.

ONE FLOW

Creado en 2015 por Adam Ruka.

- Cada nueva versión de producción está basada en la versión previa de producción.
- La mayor diferencia entre One Flow y Git Flow es que One Flow no tiene rama de desarrollo.

INSTALACIÓN I

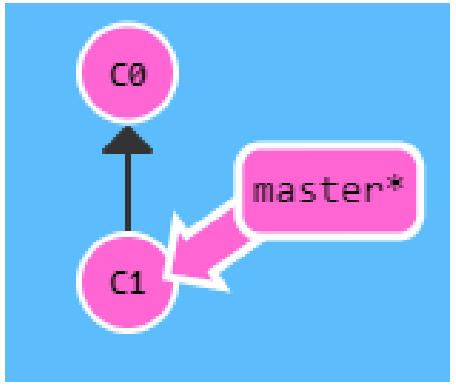
Para empezar con Git hay que descargarse el programa. Este se puede descargar desde su página oficial dónde se puede elegir la plataforma dónde lo vas a instalar.

Antes de comenzar a trabajar con Git es recomendable crearse una cuenta en Github.

Github es, según la definición en su web, una gran comunidad de programadores que crea la posibilidad de descubrir, compartir y crear mejor software. Desde proyectos de código abierto hasta repositorios de equipos privados, son la plataforma de todos para el desarrollo colaborativo. Una vez se tenga una cuenta en Github se podrá trabajar directamente con Git en tu dispositivo y sincronizar tus trabajos directamente en tu cuenta de Github.

Para empezar a aprender Git hay muchos recursos en Internet.

https://learngitbranching.js.org/?locale=es_ES



INSTALACIÓN II

Git esta planteado como un sistema multiplaforma. Existen paquetes distribuibles para los principales sistemas operativos (Linux, MacOS y Windows). Antes de descargar es conveniente comprobar, especialmente en las distribuciones Linux, si Git ya esta instalado y si es así de que versión disponemos. Para ello, desde un terminal en Linux/MacOS o una consola de Windows escribir:

```
git –versión
```

La mejor manera de instalar Git desde Linux es utilizando el gestor de paquetes de cada distribución. Así por ejemplo, en distribuciones basadas en Ubuntu (Mint, Lliurex):

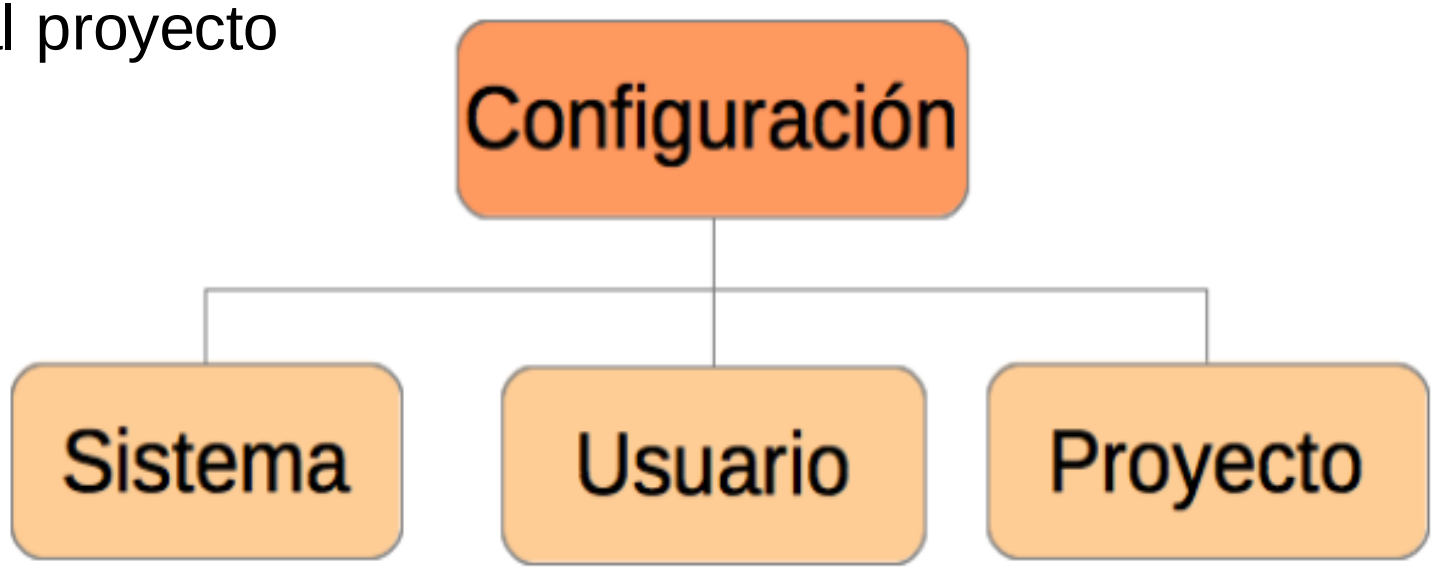
```
sudo apt-get install git
```

En diferentes versiones de Unix o Linux podemos consultar: https://git-scm.com/download/linux

CONFIGURACIÓN INICIAL I

La configuración de Git cubre tres aspectos:

- Configuración relativa al sistema general
- Configuración relativa al usuario
- Configuración relativa al proyecto



CONFIGURACIÓN INICIAL II

Cada una de estas tres configuraciones se almacena en archivos de configuración distintos de la siguiente manera:

Configuración del sistema

- Linux y MacOS fichero: /etc/gitconfig
- Windows: gitconfig en el directorio en el que se haya instalado Git

Configuración de usuario

- Linux y MacOS fichero: ~/gitconfig, es decir, en un fichero oculto dentro de la carpeta del usuario.
- Windows: .gitconfig en el directorio del usuario

Configuración del proyecto

- Dentro de la carpeta del proyecto en .git/config.

CONFIGURACIÓN INICIAL III

En un principio, si se tiene los permisos adecuados, los ficheros de configuración pueden modificarse con un simple editor de texto plano como nano, gedit o notepad. De todas maneras, para evitar posibles errores en el formato, git proporciona una herramienta para el trabajo con las configuraciones: ***git config***.
De manera general su esquema es:

```
git config [ambito] [accion] [parametro1_accion][parametro1_accion][parametro3_accion]...
```

ámbito: si la modificación es general (--system), del usuario (--global) o del proyecto (opción por defecto, no es necesario indicar nada, sólo estar en el directorio correspondiente)
acción: acción a realizar en el fichero:

- list: listar toda la configuración.
- get: obtener un valor de una variable.
- add: añadir a una variable un valor.
- unset: elimina la variable.

CONFIGURACIÓN INICIAL IV

Veamos algunos ejemplos:

- Asigna al fichero de configuración de usuario el nombre del usuario Vicent Marti. En caso de que el nombre no tuviera espacios no seria necesario incluir las comillas.

  git config --global --add user.name “Vicent Marti“

- Asigna el mail del usuario.

  git config --global --add user.email micorreo@aaaa.com

- Asigna como editor por defecto gedit

  git config --global core.editor gedit

- Elimina del fichero de configuración de usuario la variable email de la sección user.

  git config --global --unset user.email

- Muestra en pantalla las variables definidas en todos los ficheros (general, usuario y sistema)

  git config --list

EJERCICIO 2

Modifica los ficheros de configuración general y de usuario de tal manera que:

- Configuración de usuario: nombre e email del usuario
- Configuración del sistema: editor de texto por defecto (gedit en Linux, notepad en Windows y open -e en MacOS)
- ¿Qué comando utilizarías para volver al editor por defecto de git?

Entrega

- Ficheros de configuración general y de usuario.
- Escribe una documento con portada, índice, url's consultadas, etc., con la captura de pantalla con la lista de todas configuraciones y los comandos utilizados.
- Súbelo al aula virtual (AV).