.UBAfiuba
FACULTAD DE INGENIERÍA

MIPS_SQUAD

Integrantes

- Melina Jauregui
- Ian Gauler
- Patricio Tourne Passarino
- Sofía Gómez Belis

Introducción

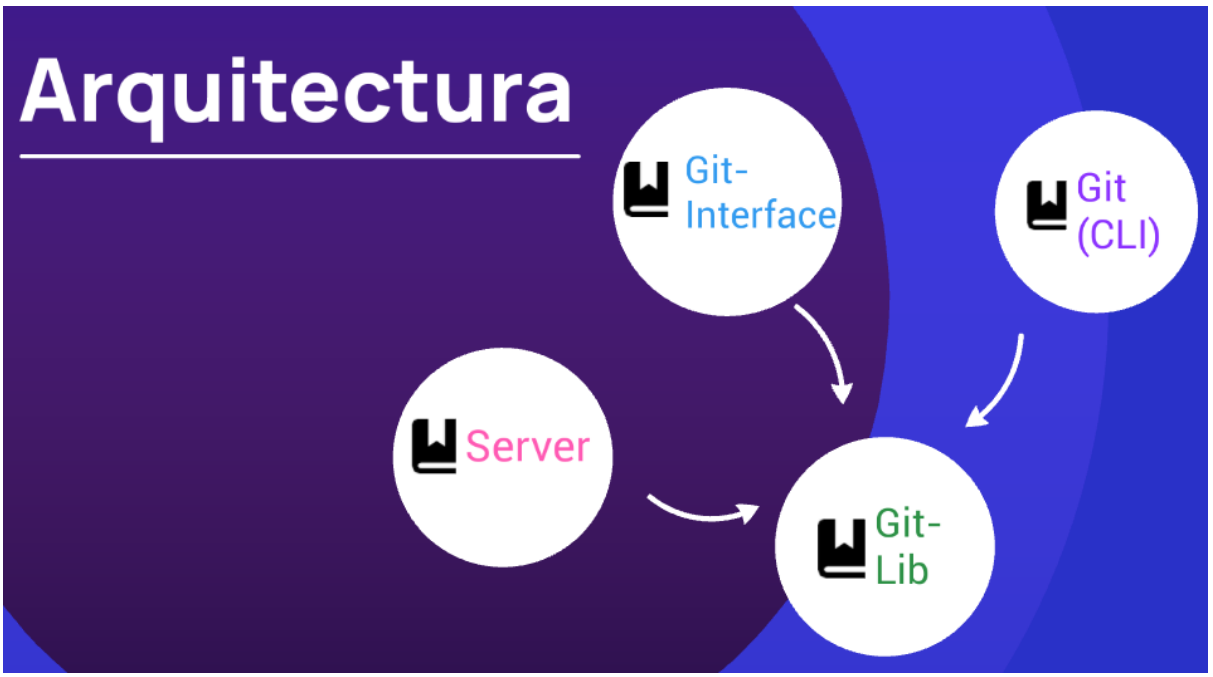
En el desarrollo del proyecto de implementación del Cliente y Servidor Git, el objetivo principal ha sido proporcionar una funcionalidad básica que permita a los usuarios interactuar con repositorios y realizar operaciones clave. Este trabajo se centra en una selección de comandos fundamentales de Git, sin abordar en detalle aspectos más avanzados o personalizados.

Desarrollo

Arquitectura

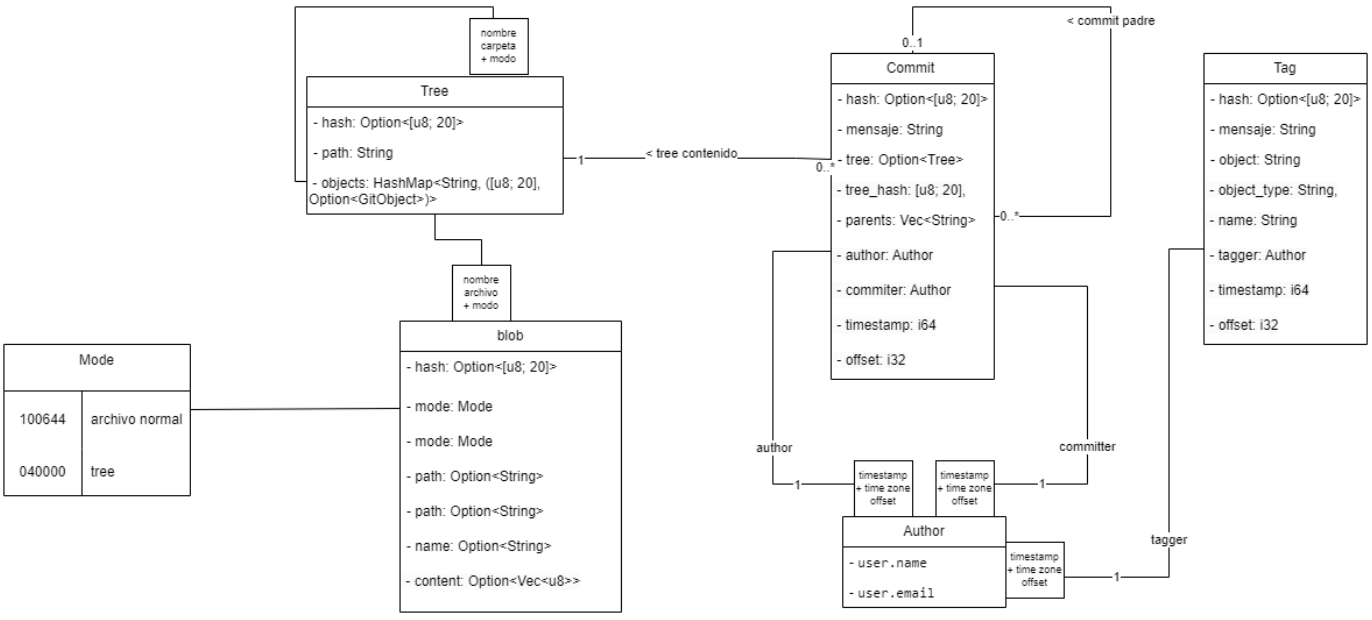
Nuestro proyecto está compuesto por cuatro módulos: git, git-lib, git-interface y server.

- Git → Es la CLI. Contiene parte de la implementación de los comandos de git, como el manejo de flags y errores.
- Git-lib → Es la biblioteca del proyecto, utilizada por los otros componentes de la arquitectura.
    - GitRepository → Representa el repositorio. Todos los comandos lo utilizan para ejecutarse.
    - CommandErrors → Incluye la lista de errores admitidos por nuestra implementación.
    - GitObject → Representa un objeto git.
    - ObjectsDatabase → Representa la base de datos del repositorio. Se utiliza para leer y guardar objetos git.
    - StagingArea → Desde el repositorio, se puede abrir y guardar el staging area, así como agregar o eliminar archivos del mismo.
    - ChangesController → Es una estructura utilizada por múltiples comandos y la interfaz para obtener los archivos que están en el staging area, los no registrados por git y los cambios sin confirmar. Permite saber el tipo de modificación de un archivo.
- Git-interface → Es la GUI.
- Server → Incluye todo el código relacionado al servidor.



Git Objects

Uno de los focos principales de nuestra implementación son los objetos de git: blob, tree, tag y commit. El trait GitObject incluye funcionalidades comunes a todos los objetos. Los structs Blob, Tree, Tag y Commit lo implementan y agregan operaciones adicionales utilizadas en todo el proyecto.



Servidor

Se creó una estructura llamada server que modela el servidor git con nuestra implementación. Este implementa los protocolos update-pack y receive-pack.

El protocolo upload-pack permite al cliente descargar el contenido de un repositorio. Cuando un Cliente Git desea clonar un repositorio o realizar una operación que requiera obtener información desde el servidor remoto, como fetch, inicia una solicitud upload-pack. El servidor la procesa y envía al cliente la información necesaria, que incluye referencias a las ramas, confirmaciones (commits) y objetos necesarios para la operación.

Por otro lado, el protocolo receive-pack permite al cliente enviar actualizaciones al servidor, como confirmaciones y otros objetos. Cuando un cliente realiza una acción que implica enviar cambios al repositorio remoto, como push, inicia una solicitud receive-pack. El servidor la procesa y actualiza su copia del repositorio con los cambios enviados por el cliente.

Para cada protocolo se hicieron sus respectivas funciones que modelan la negociación. Además se crearon funciones auxiliares como:

- check_commits_between: verifica que estén todos los objetos en el packfile entre el nuevo y el viejo commit al que apunta cada rama.
- send: manda mensajes al socket
- build_packfile : arma el packfile siguiendo el protocolo git

Delta Objects:

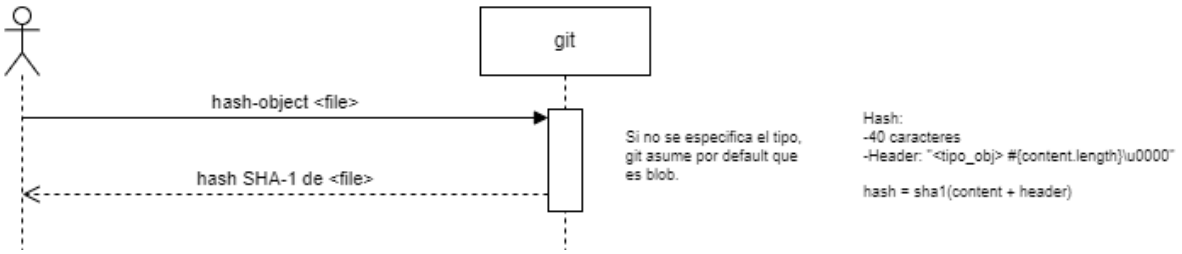
Implementamos los dos tipos posible de Delta Objects, los cuales son:

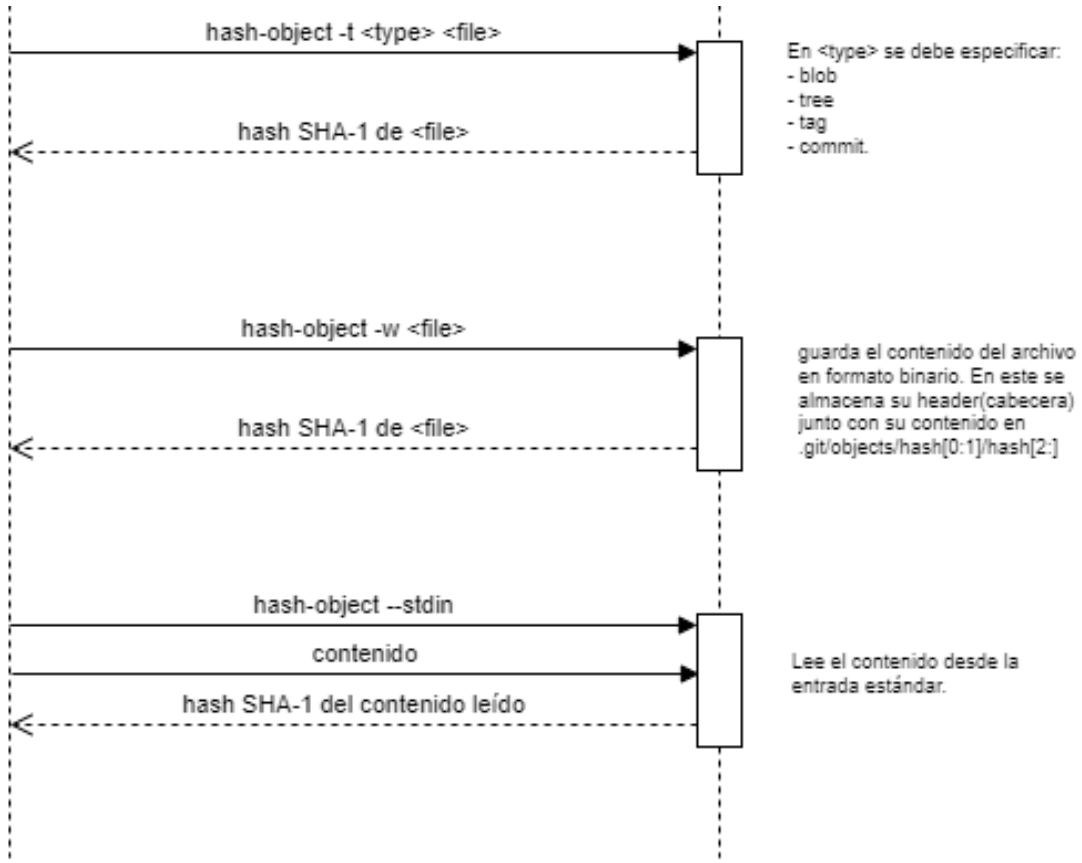
- OfsDelta
- RefDelta

De esta forma, a través de la utilización de funciones como read_objects_from_packfile_body y read_objects_from_offset, se hizo posible leer los objetos del tipo Delta en los packfiles. Estas funciones se encargan de poder obtener las instrucciones de cada delta y los objetos bases de estos.

Hash-object

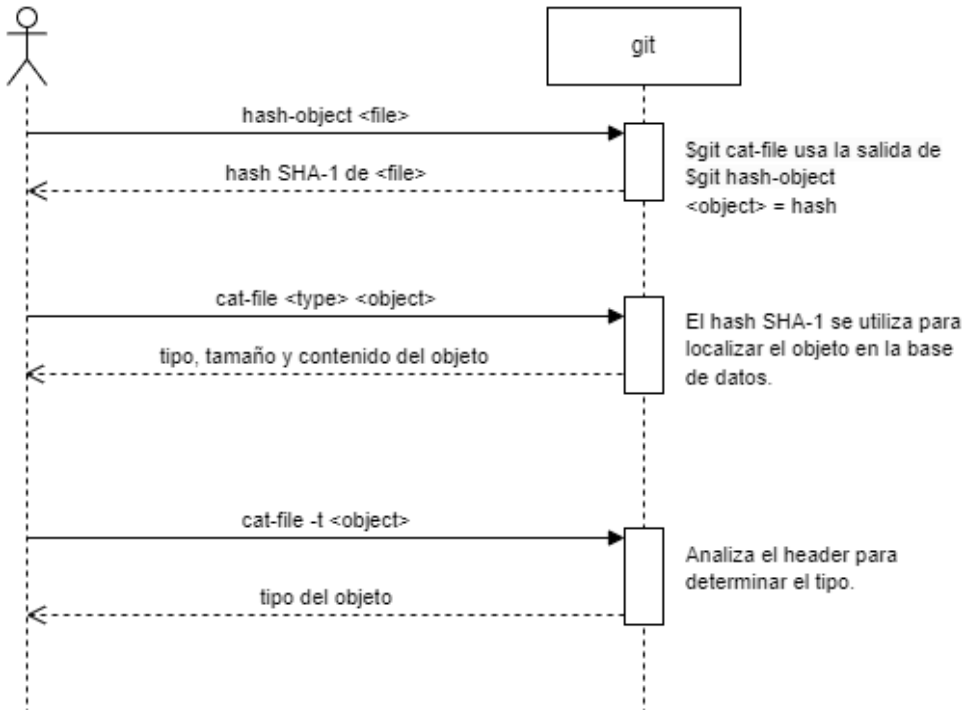
Calcula el sha-1 de un objeto dado su contenido. Opcionalmente, puede escribir el objeto resultante en la base de datos. Utilizamos el trait GitObject para obtener el hash. Los objetos conocen su contenido. Por lo tanto, les pedimos directamente que calculen su hash.

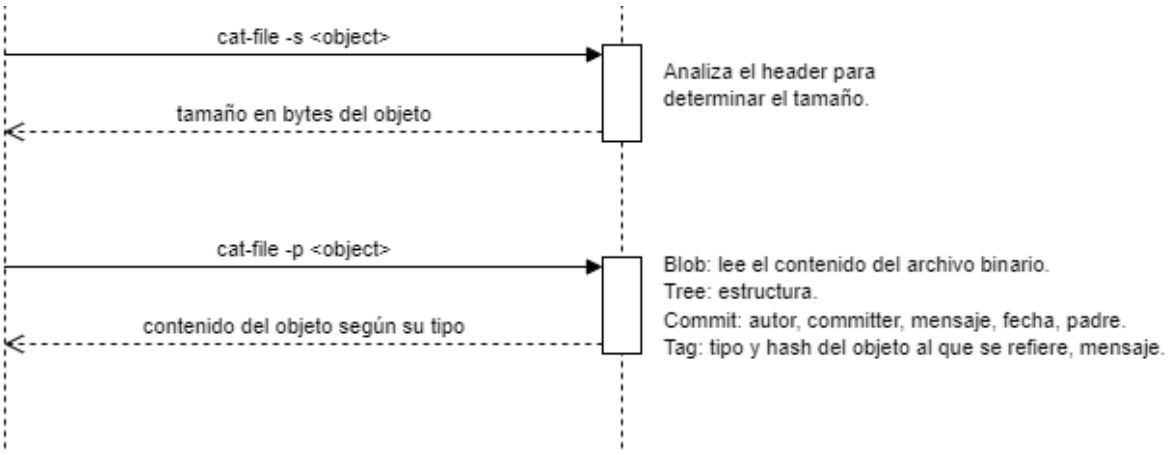




Cat-file

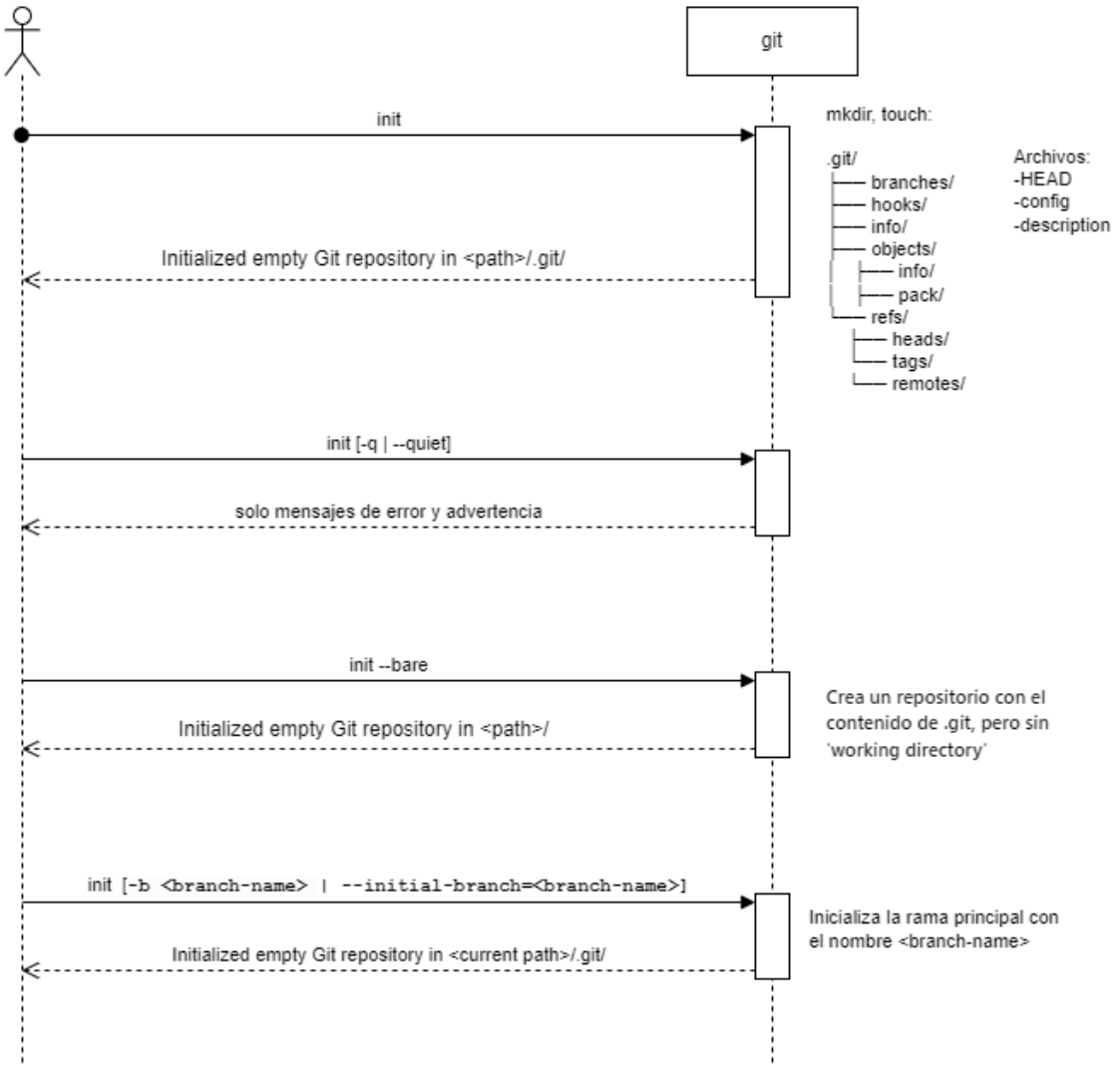
Cat-file nos permite obtener información de un objeto git a partir de su hash. Esto incluye tipo, tamaño y contenido.





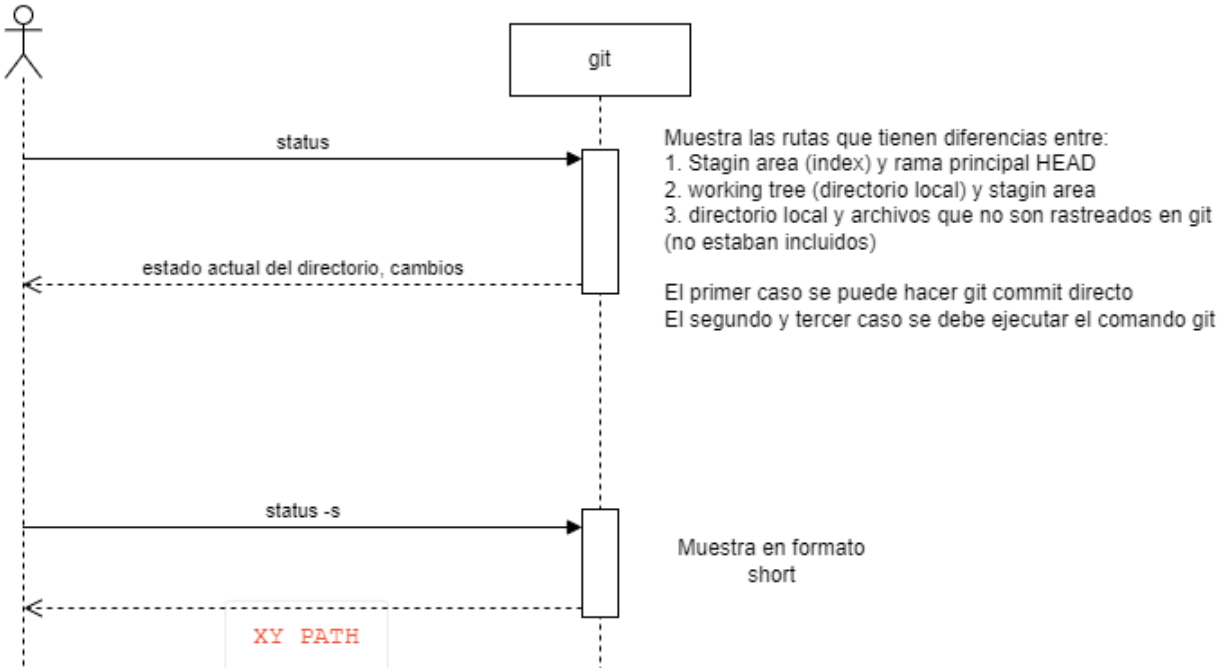
Init

Init es el comando utilizado para crear un repositorio Git vacío o reinicializar uno existente. En el primer caso, se crea el directorio .git con los subdirectorios y archivos correspondientes, como .git/objects/, .git/refs/, .git/HEAD, entre otros. Además, crea la rama principal del repositorio, comúnmente denominada 'master' o 'main'.



Status

El comando ‘status’ muestra los archivos que tienen diferencias entre su estado en el índice y el último commit de la rama actual (Changes to be commited), archivos cuyo estado en el índice y el directorio de trabajo difieren (Changes not staged for commit), y archivos que no están siendo rastreados por Git (untracked paths). Durante un merge, indica los archivos que tienen conflictos.



Utilizamos la estructura auxiliar ChangesController para implementar el funcionamiento del comando. Su objetivo es registrar todos los archivos que cumplen con las condiciones mencionadas y el tipo de cambio, ya sea agregado, modificado o eliminado.

Por defecto, status muestra su salida en formato ‘long’, listando en orden:

- Opcional → archivos con conflictos
- Cambios a ser confirmados → Se encuentran guardados en el staging area, pero aún no han sido confirmados.
- Cambios no guardados → Son cambios en el directorio de trabajo que no se han guardado en el index.
- Archivos no rastreados → Son archivos que nunca se han agregado al index.

Agregamos también la posibilidad de mostrar el estado del directorio de trabajo y el index en formato ‘short’

```
X          Y     Meaning
-------------------------------------------------
     [AMD]   not updated
M        [ MTD]  updated in index
T        [ MTD]  type changed in index
A        [ MTD]  added to index
D                deleted from index
R        [ MTD]  renamed in index
C        [ MTD]  copied in index
[MTARC]          index and work tree matches
[ MTARC]     M    work tree changed since index
[ MTARC]     T    type changed in work tree since index
[ MTARC]     D    deleted in work tree
        R    renamed in work tree
        C    copied in work tree
-------------------------------------------------
D           D    unmerged, both deleted
A           U    unmerged, added by us
U           D    unmerged, deleted by them
U           A    unmerged, added by them
D           U    unmerged, deleted by us
A           A    unmerged, both added
U           U    unmerged, both modified
-------------------------------------------------
?           ?    untracked
!           !    ignored
-------------------------------------------------
```

Formato short

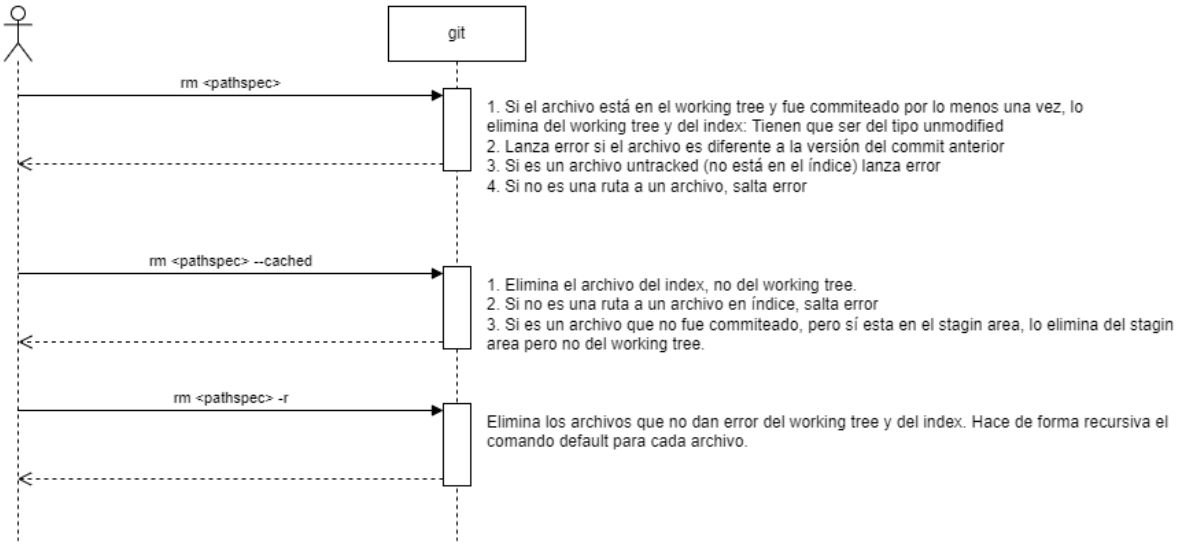
Utilizamos la estructura Format para mostrar el resultado del comando.

Add

Git ‘add’ es uno de los comandos que interactúa directamente con el índice, agregando o actualizando los archivos guardados en el mismo, con su contenido en el directorio de trabajo. Su objetivo es preparar el staging area para el siguiente commit. Nuestra implementación no soporta el flag –force, por lo cual los archivos ignorados no serán agregados al índice. En cambio, se pueden agregar archivos determinados, todos los componentes de un directorio o todos los cambios nuevos.

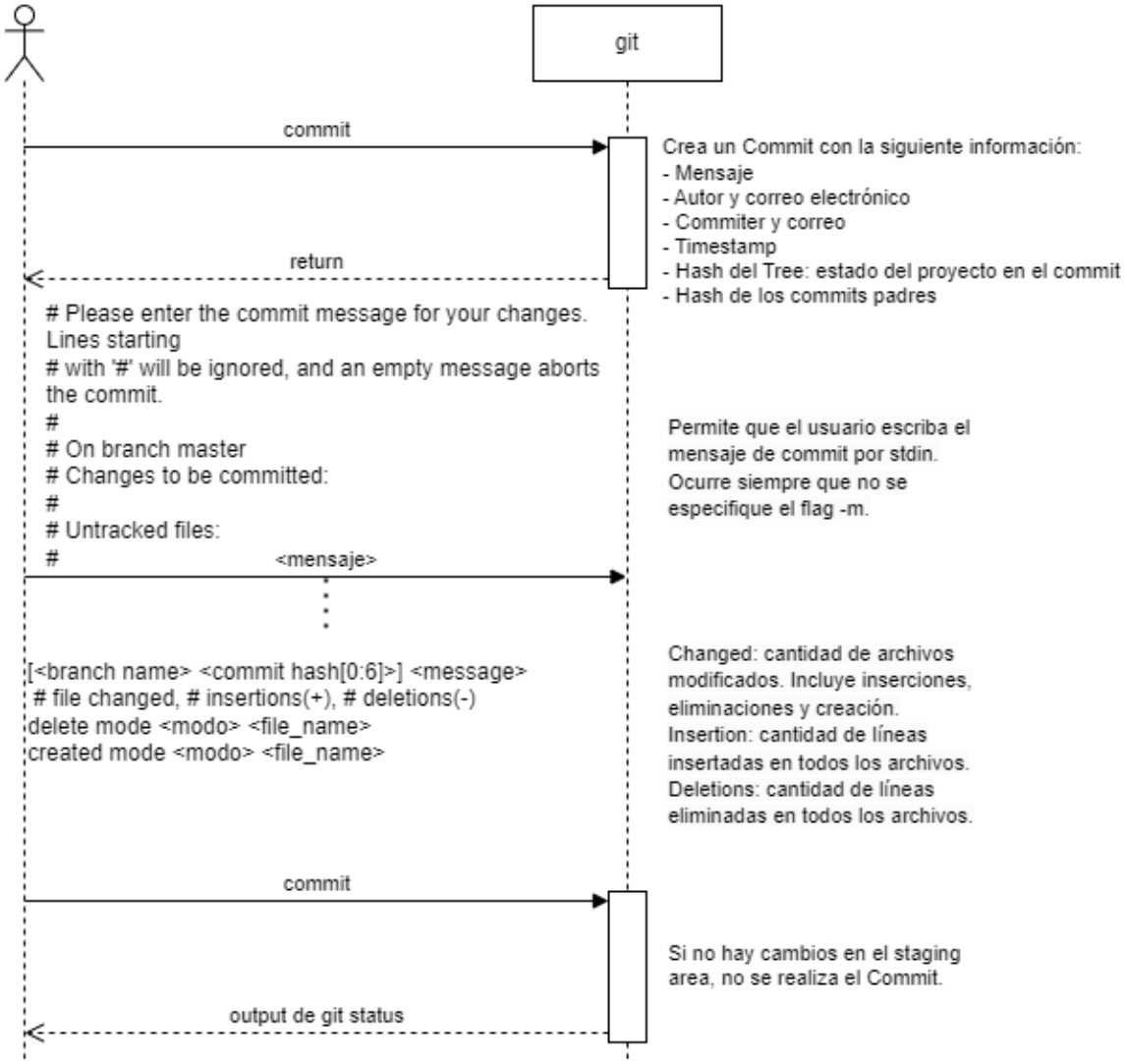
Rm

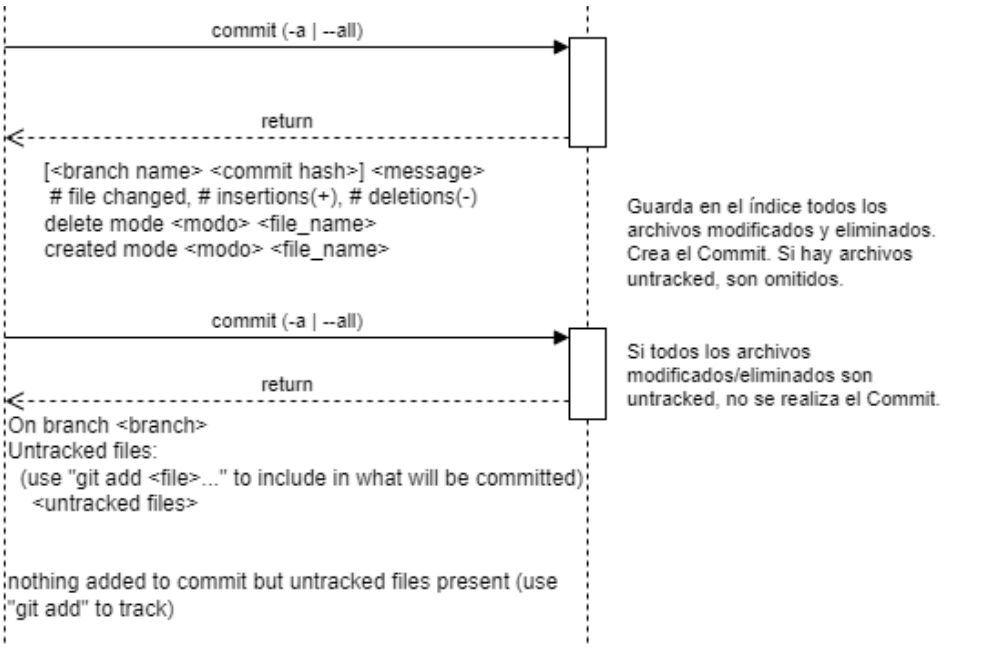
Elimina archivos del directorio de trabajo y el índice, o únicamente del índice. Los archivos que se eliminan deben ser idénticos a su versión en el commit anterior.

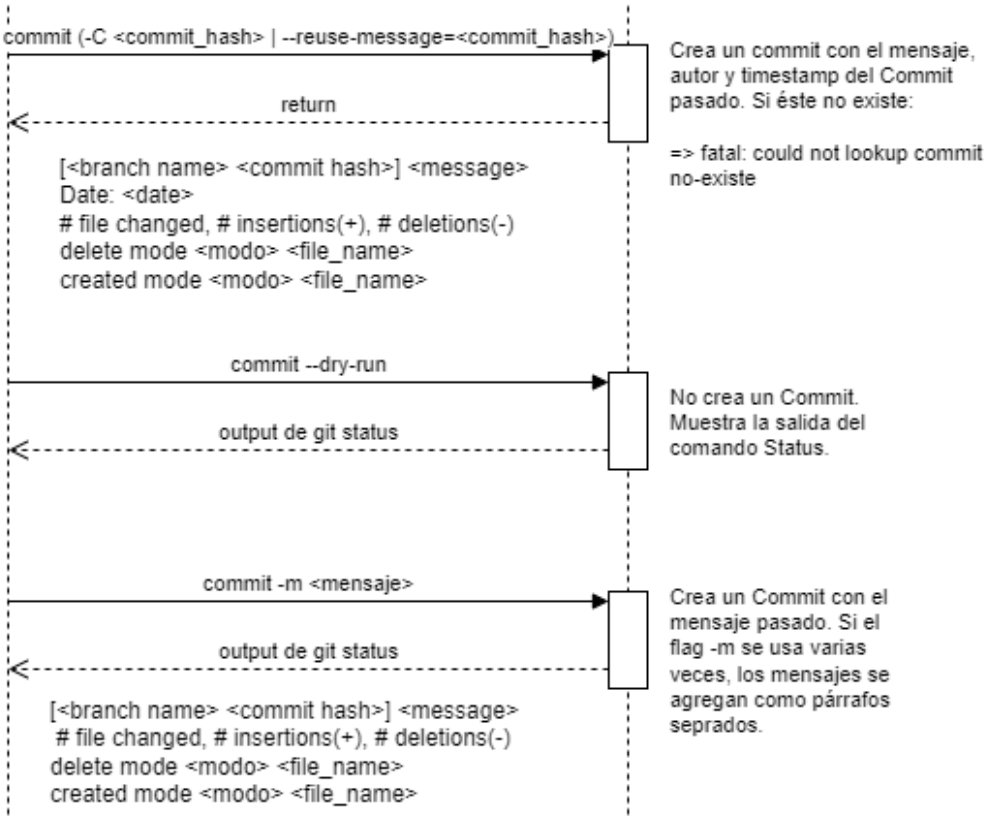


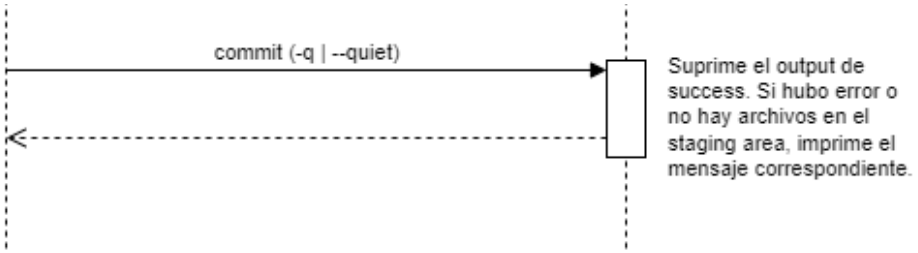
Commit

El comando 'commit' crea y guarda en la base de datos un objeto Commit que contiene el contenido actual del índice y el mensaje proporcionado. Su padre es el último commit de la rama actual, la cual se actualiza para apuntar a ella.



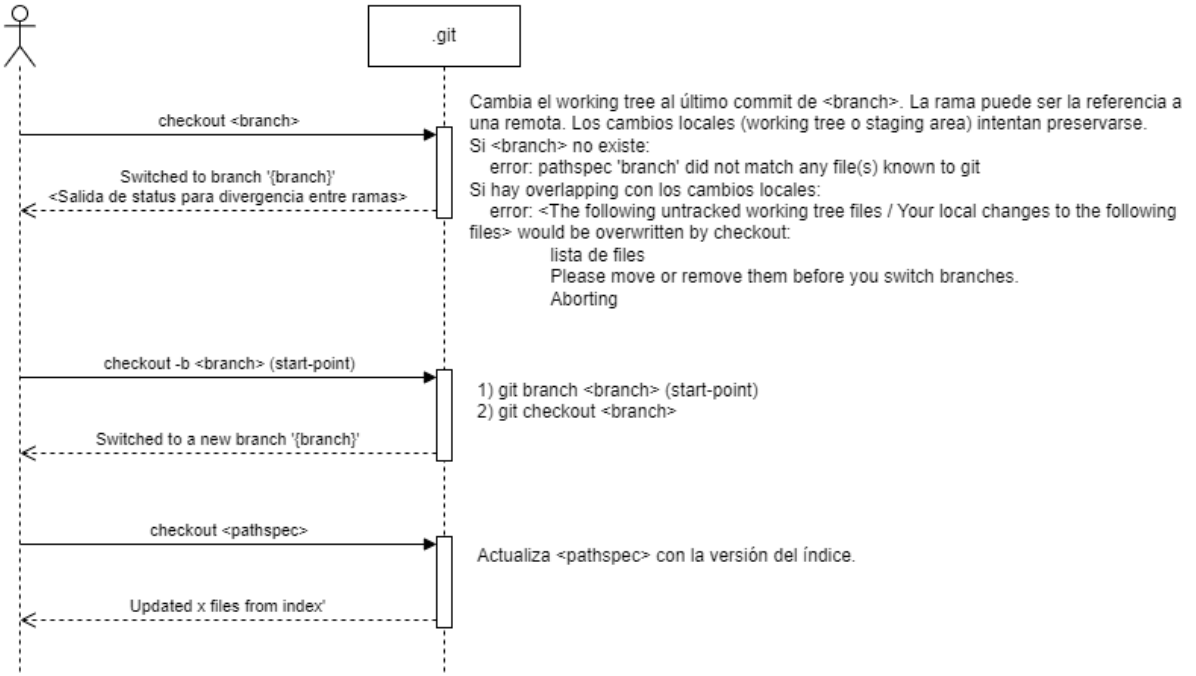

commit (-a | --all)
return
[<branch name> <commit hash>] <message>
file changed, # insertions(+), # deletions(-)
delete mode <modo> <file_name>
created mode <modo> <file_name>
Guarda en el índice todos los archivos modificados y eliminados. Crea el Commit. Si hay archivos untracked, son omitidos.
commit (-a | --all)
return
Si todos los archivos modificados/eliminados son untracked, no se realiza el Commit.
On branch <branch>
Untracked files:
(use "git add <file>..." to include in what will be committed)
<untracked files>
nothing added to commit but untracked files present (use "git add" to track)


commit (-C <commit_hash> | --reuse-message=<commit_hash>)
return
Crea un commit con el mensaje, autor y timestamp del Commit pasado. Si éste no existe:
=> fatal: could not lookup commit no-existe
[<branch name> <commit hash>] <message>
Date: <date>
file changed, # insertions(+), # deletions(-)
delete mode <modo> <file_name>
created mode <modo> <file_name>
commit --dry-run
output de git status
No crea un Commit. Muestra la salida del comando Status.
commit -m <mensaje>
output de git status
Crea un Commit con el mensaje pasado. Si el flag -m se usa varias veces, los mensajes se agregan como párrafos seprados.
[<branch name> <commit hash>] <message>
file changed, # insertions(+), # deletions(-)
delete mode <modo> <file_name>
created mode <modo> <file_name>


commit (-q | --quiet)
Suprime el output de success. Si hubo error o no hay archivos en el staging area, imprime el mensaje correspondiente.

Checkout

Este comando es utilizado para cambiar de rama. También permite crear una nueva antes de hacer el checkout. Para ello, utiliza nuestra implementación de 'git branch'. Checkout actualiza archivos en el directorio de trabajo para que coincidan con la versión en el índice de la rama especificada. Las modificaciones de los archivos en el directorio actual intentan conservarse, a menos que se encuentren conflictos que impidan el cambio de rama. Nuestra implementación también permite actualizar archivos con su versión en el índice, sin realizar cambio de rama.



Log

Log es un comando que obtiene el historial de commits realizados desde el commit en el que estamos trabajando, excepto que se use la flag –all, que indica que se imprimirá el historial de commits desde el commit más reciente de cada rama de trabajo.

Esto se logra obteniendo el padre de cada commit, y así sucesivamente hasta llegar al commit inicial.

Al comando le agregamos una funcionalidad adicional, que es la de obtener de qué rama proviene cada commit, para su uso posterior en nuestro grafo utilizado en la interfaz. El problema de esta funcionalidad radica cuando necesitamos obtener el padre de un merge commit, debido a que existe una bifurcación. Esto lo solucionamos de manera sencilla debido a que git guarda en la base de datos el commit al que se aplicaron los cambios de merge antes del commit que se obtuvo los cambios pertinentes. Por lo tanto, sabemos que el primer commit obtenido de la base de datos proviene de la rama de la que veníamos obteniendo aguas arriba, y los commits siguientes en la base de datos son aquellos que dan sus cambios.

Clone

El comando Clone replica un repositorio remoto de manera precisa. Lo hemos diseñado para depender únicamente del comando pull, el cual a su vez se apoya en los comandos fetch y merge, ejecutándolos en ese orden respectivamente.

Fetch

Obtiene ramas y/o referencias del repositorio, así como con los objetos necesarios para completar el historial. Descarga los cambios del repositorio remoto, pero no los aplica automáticamente. Las referencias de las ramas remotas son actualizadas.

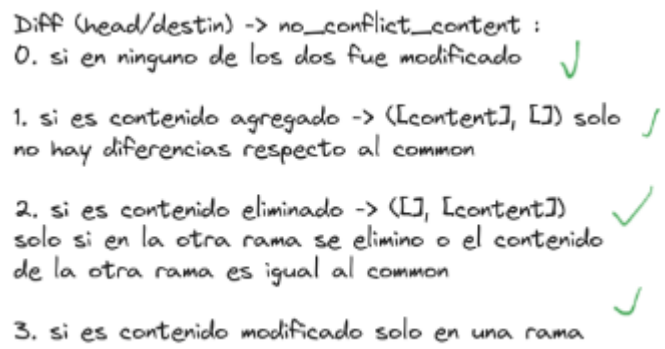
Los nombres de las referencias que se obtienen, junto con los de los objetos a los que apuntan, se escriben en .git/FETCH_HEAD.

Para realizar estas operaciones, nuestro cliente Git se comunica con el Servidor implementado, que se utiliza primero para actualizar las ramas remotas y posteriormente guardar los objetos. Se hace uso del protocolo update-pack.

Merge

Git merge es un comando que fusiona cambios de una rama en otra. Se utiliza para combinar el historial de dos ramas diferentes, incorporando las modificaciones de una rama en la otra. Esto puede generar posteriormente un nuevo commit de fusión si las ramas tienen cambios que no se pueden aplicar de manera automática. Nuestra implementación incluye el 'merge continue' para terminar el merge luego de la resolución de conflictos.

Dados los contenidos de los commits a fusionar y el ancestro común, los comparamos con el objetivo de encontrar conflictos de merge. A continuación indicamos cuándo consideramos que no hay conflicto:



Pull

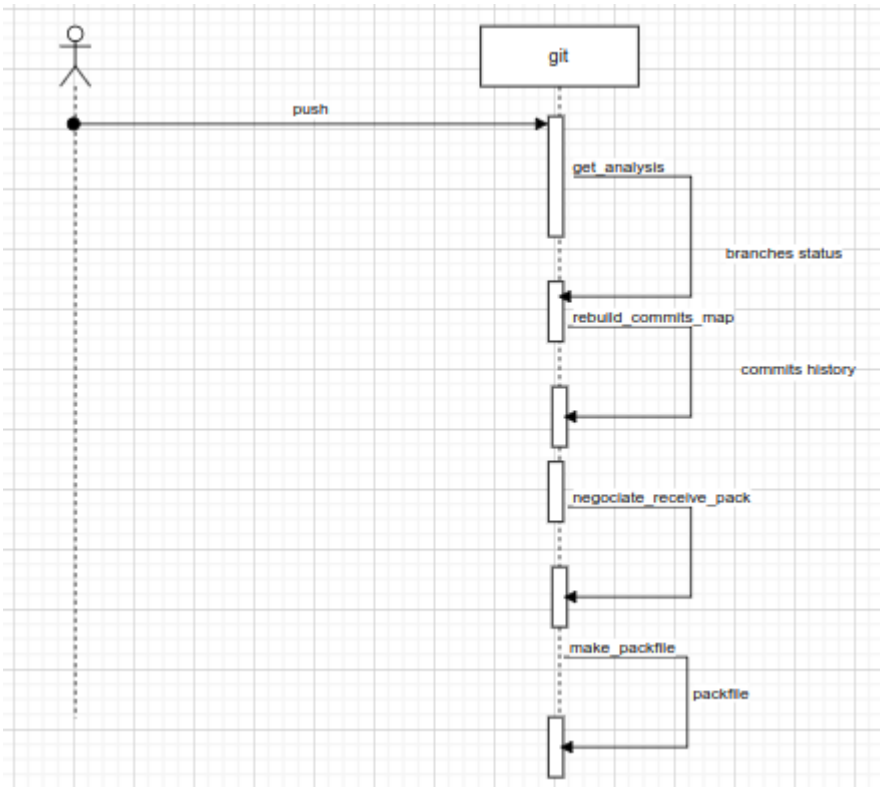
Incorpora los cambios del repositorio remoto a la rama actual. Realiza dos operaciones consecutivas, fetch y merge. Si hay conflictos, se deberán resolver manualmente. Si alguno de los cambios remotos se superpone con cambios locales no confirmados, la fusión se cancelará automáticamente y el directorio de trabajo permanecerá intacto.

Push

Actualiza las referencias de las ramas remotas en base a las locales y envía los objetos necesarios para completarlas.

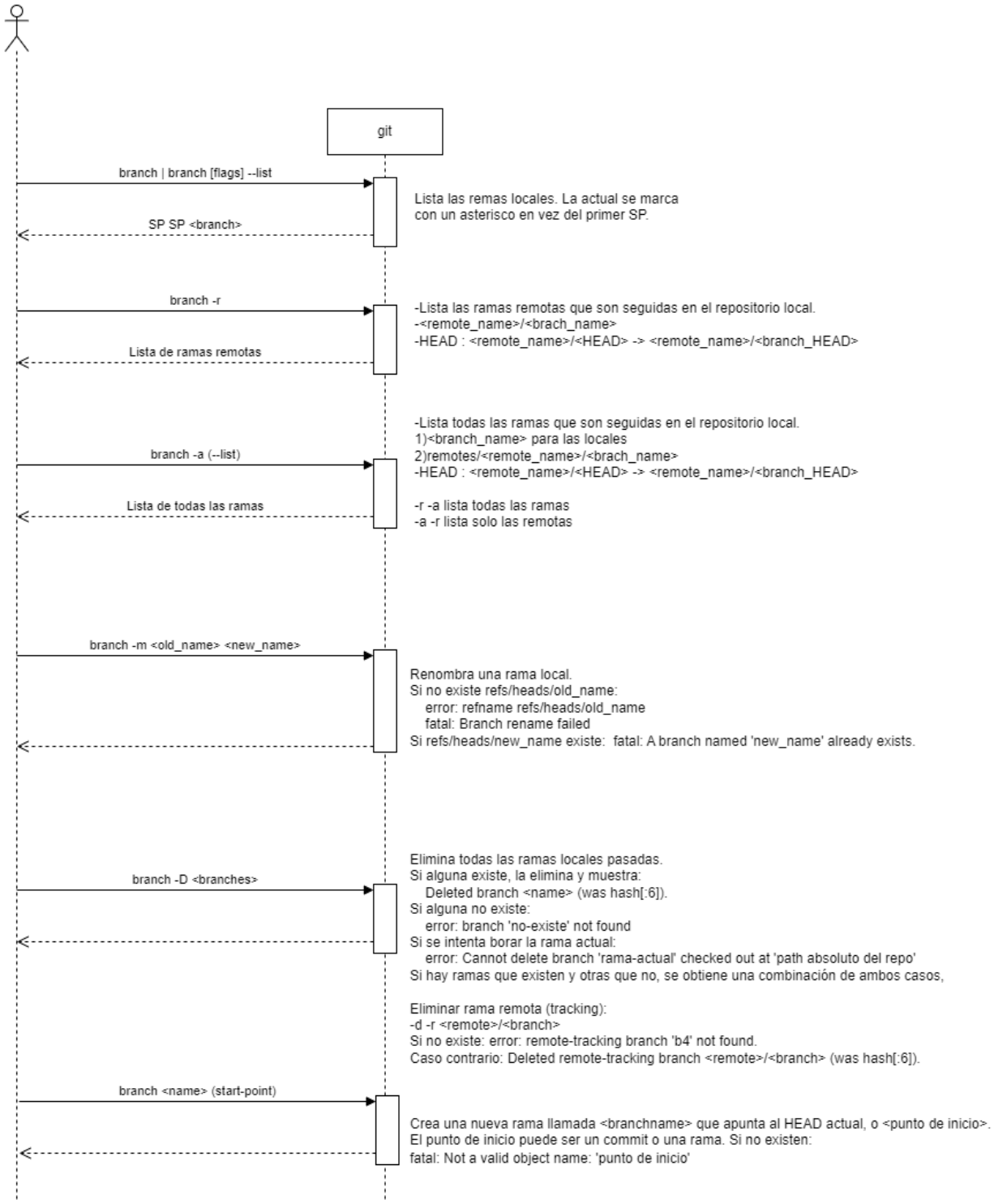
El comando Push ejecuta el protocolo receive-pack en el servidor. Este utiliza dos funciones fundamentales para poder seguir con el protocolo las cuales son:

- get_analysis : función auxiliar llamada utilizada para obtener la información recibida del servidor. Esta hace a su vez la recopilación del estado de las branches en este.
- rebuilds_commit_tree : función auxiliar que recopila todos los commits desde que se llamo por primera vez hasta encontrar un commit en el vector pasado por parámetro hash_to_look_for.
- negociate_receive_pack : Es la función que le indica al servidor qué ramas debe actualizar luego de que finalizara la negociación.
- make_packfile : utilizando el commits_map crea el packfile para que luego sea enviado hacia el servidor.



Branch

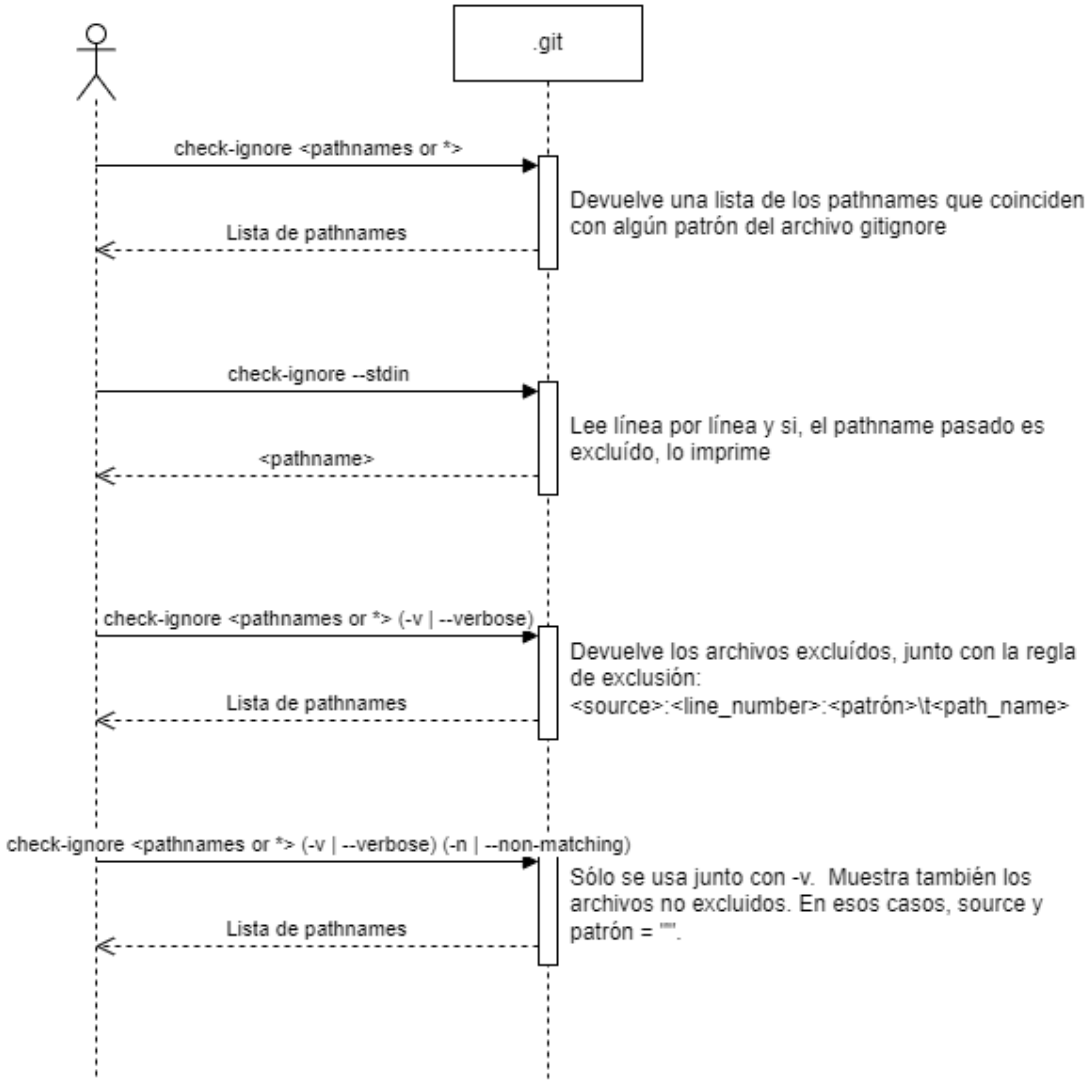
El comando ‘branch’ tiene múltiples funcionalidades, de las cuales implementamos la creación, cambio de nombre y eliminación de ramas, así como la posibilidad de listar tanto ramas locales como remotas. Por default, las ramas se crean a partir del último commit de la rama actual. Sin embargo, es posible indicar un punto de inicio, como otra rama o commit.



Check-ignore

Check-ignore revisa todos los archivos de exclusión del repositorio, es decir, .git/info/exclude y los .gitignore. Para cada ruta proporcionada a través de la línea de comandos o desde --stdin, verifica si coincide con alguno de los patrones hallados, en cuyo caso la muestra.

Los archivos ignorados por git no se incluyen en commits ni son mostrados por defecto en ‘status’.



La estructura GitignorePatterns analiza los siguientes patrones soportados por nuestra implementación:

- *name.txt → El asterisco puede ser reemplazado por varios caracteres. Todos las rutas que terminen con 'name.txt' serán ignoradas
- /name* → Los archivos o directorios que empiecen con 'name' serán ignorados.
- name → Es un patrón no relativo. Esto implica, por ejemplo, que a/name también será ignorado.
- name/a → Los patrones con barras al principio o al medio son relativos al directorio actual. En este caso, a/name/a no será ignorado.
- !patrón → Ignora el patrón dado.

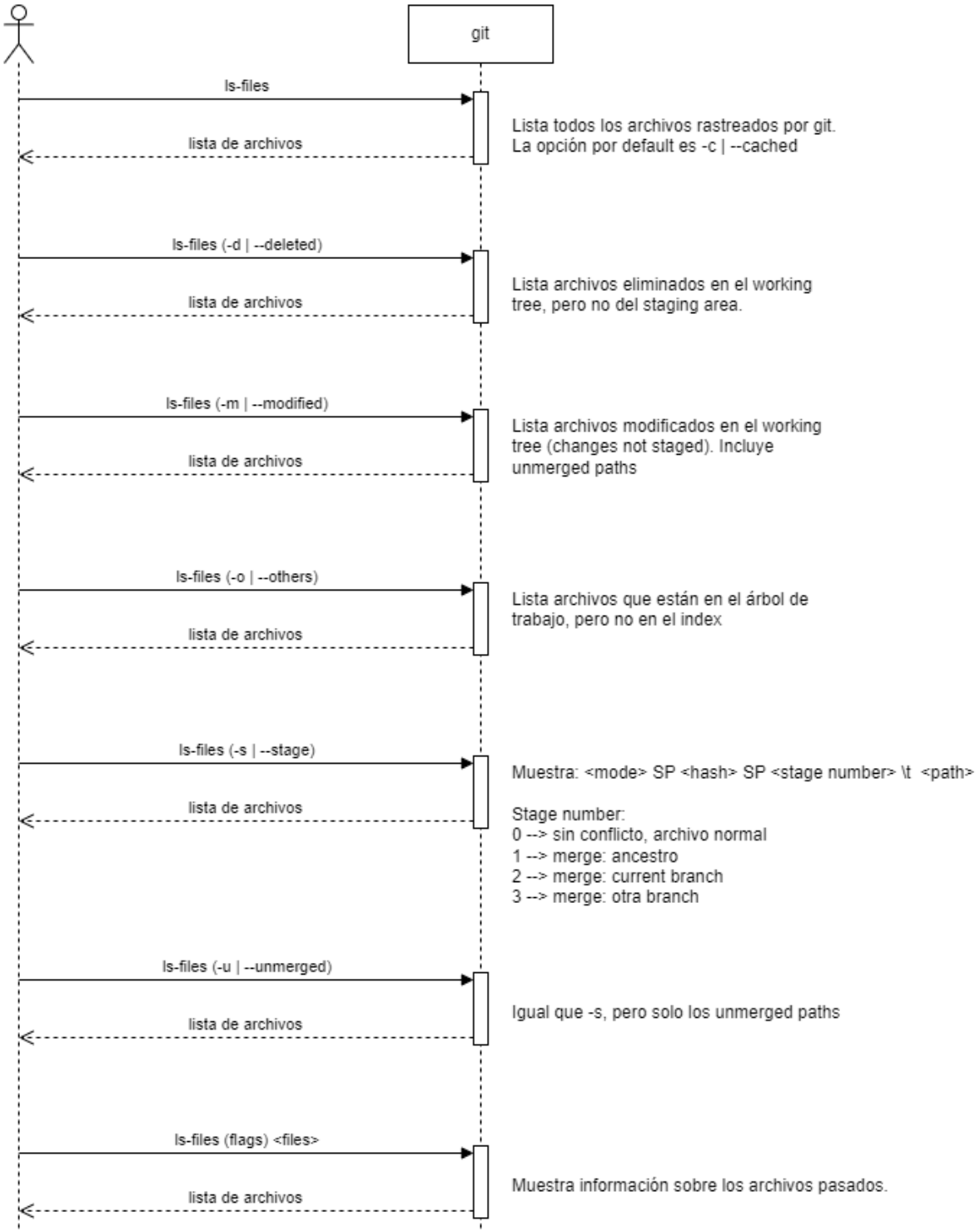
Si una ruta cumple un patrón dado, el mismo será, obtenido por orden de precedencia, de mayor a menor:

1. .gitignore
2. .git/info/exclude

Dentro del mismo archivo de exclusión, se considera el último patrón coincidente.

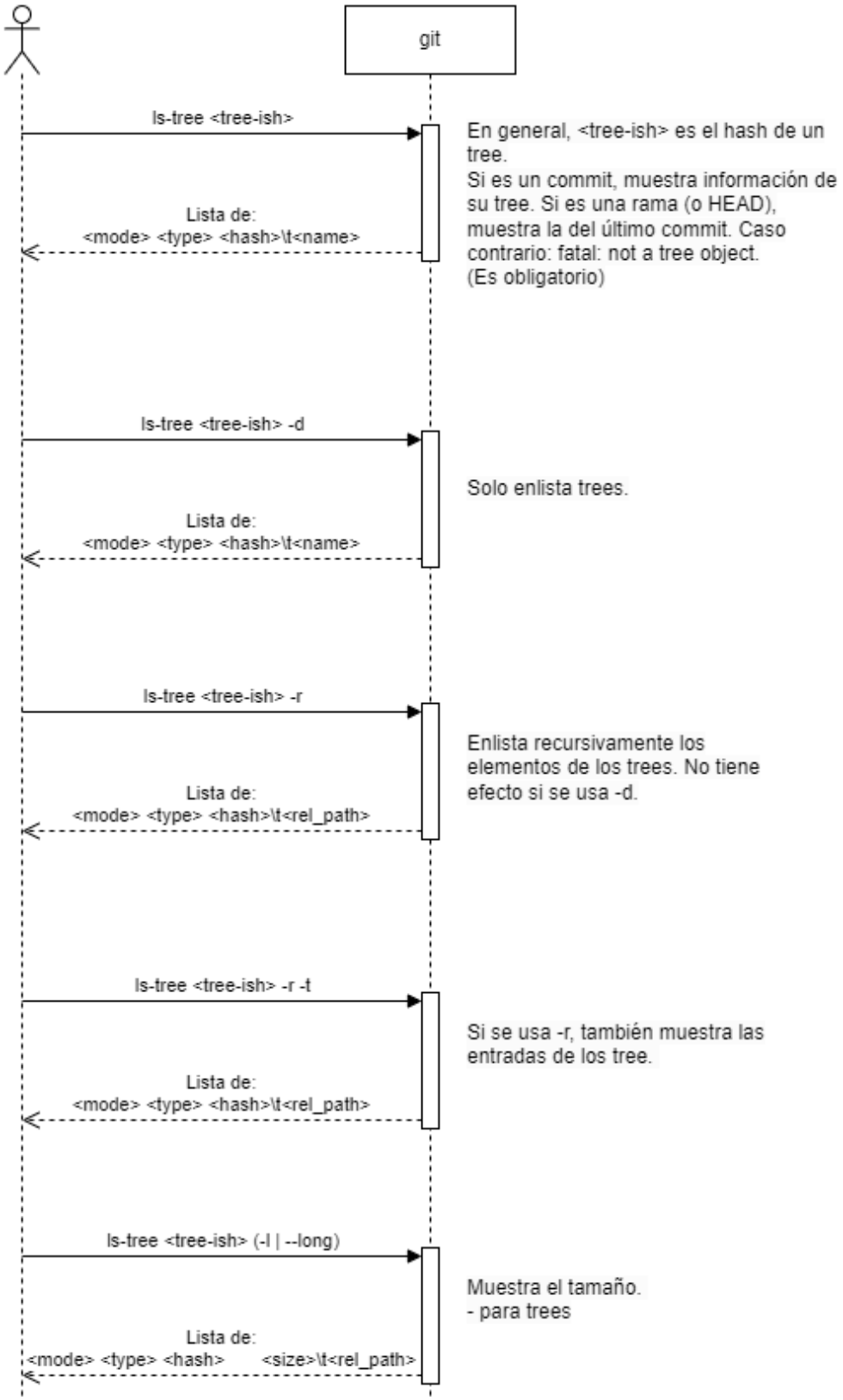
Ls-files

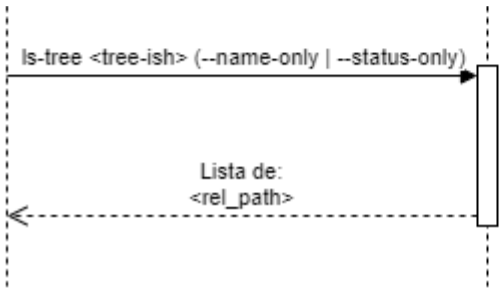
En general este comando es utilizado para listar los archivos rastreados por git, pero opcionalmente permite postrar aquellos que git no conoce, que tienen conflictos de merge o incluso los que se han eliminado. Además, puede proporcionar información extra de estos archivos.



Ls-tree

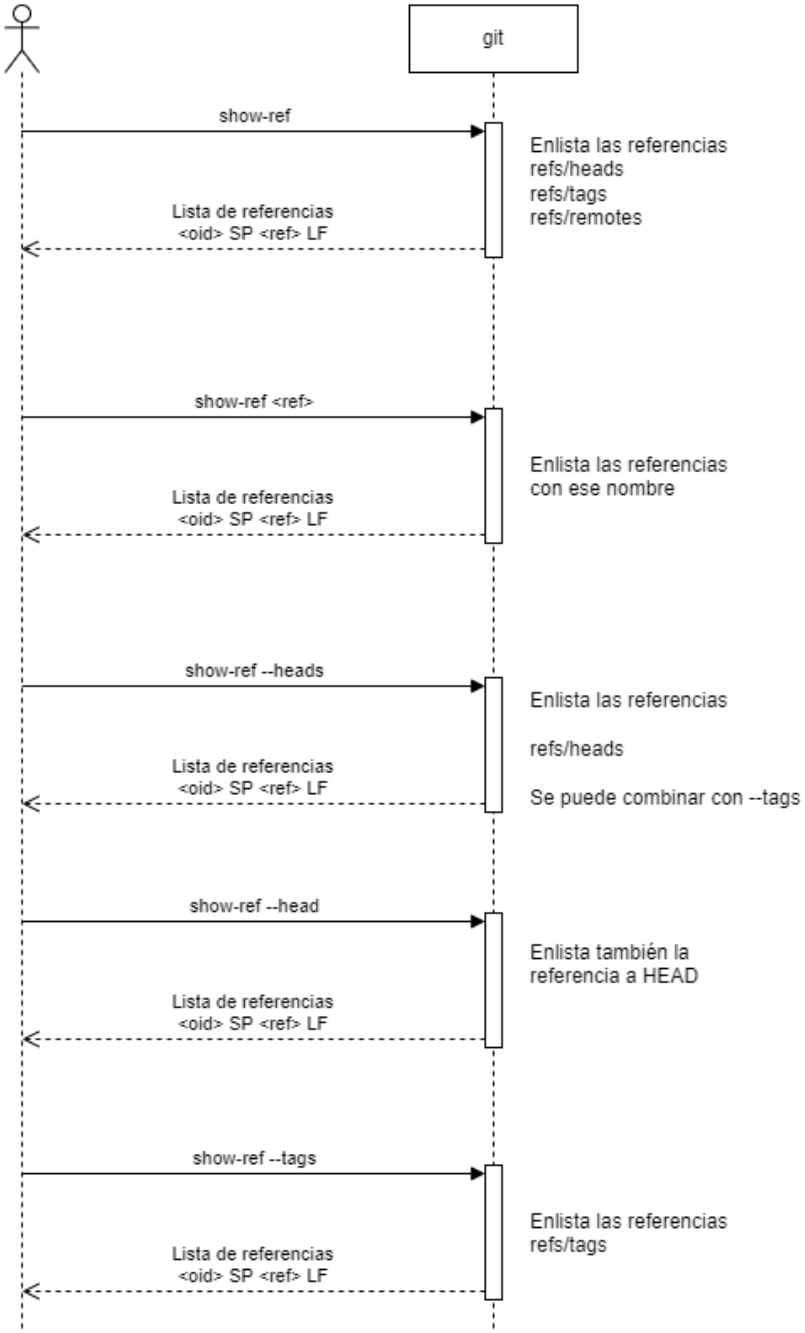
Muestra el contenido de un árbol, pudiendo ser también de forma recursiva. El comando recibe un hash que puede referirse a un tree o a un commit, o el nombre de una rama. En los últimos casos, devuelve la información del árbol asociado.

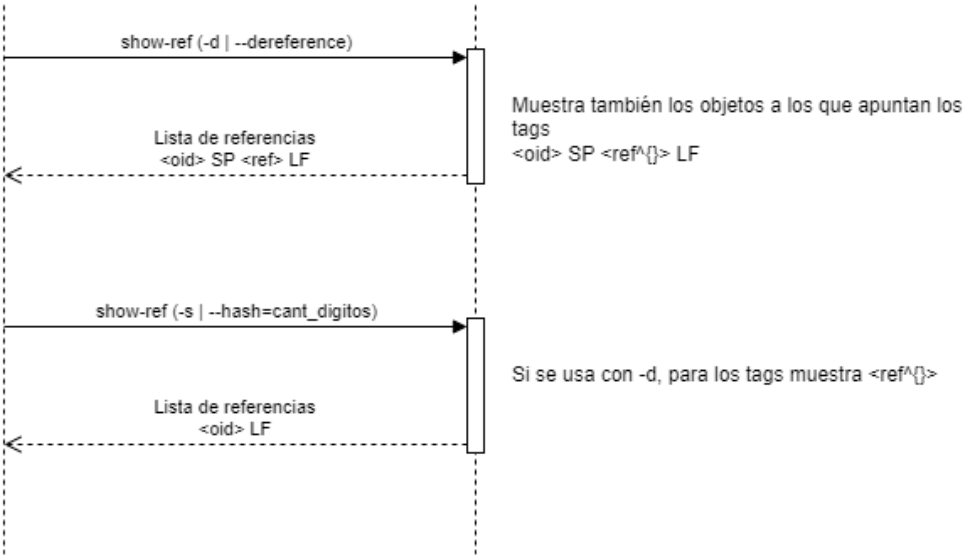




Show-ref

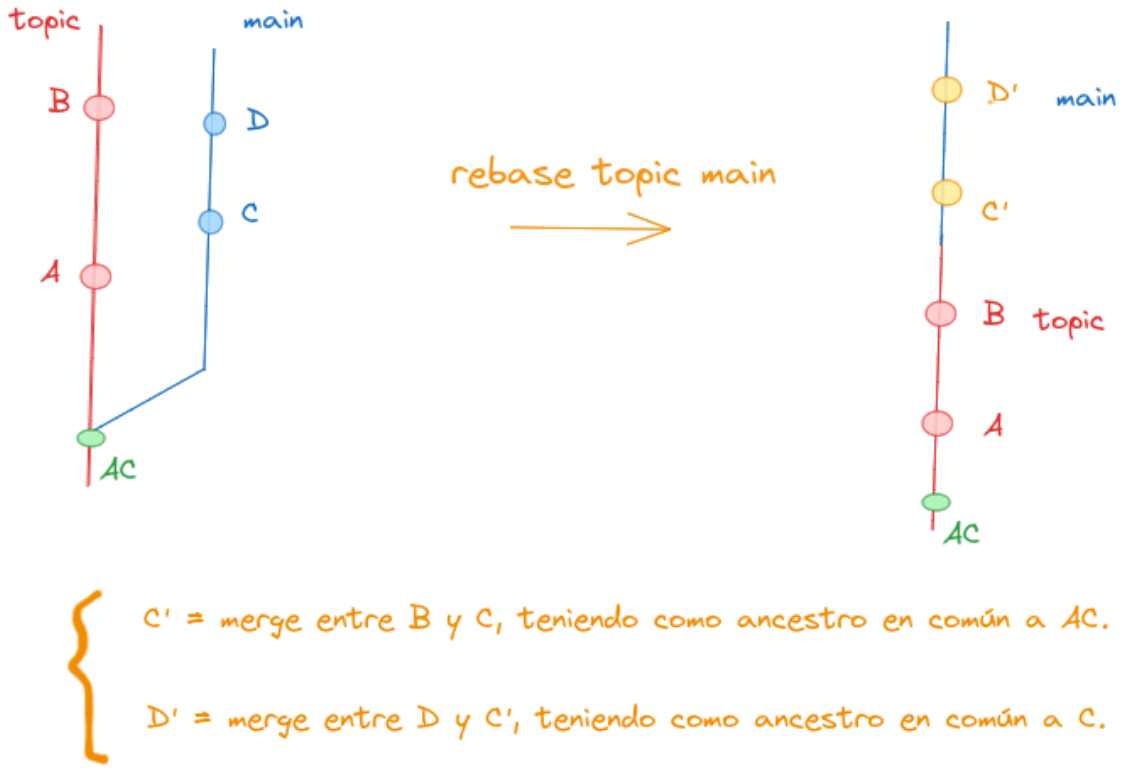
Se utiliza para listar las referencias del repositorio. Esto incluye los archivos en .git/refs/heads, .git/refs/remotes y .git/refs/tags.





Rebase:

El comando rebase se implementó de tal forma que utiliza dos estructuras Hash para poder llevar a cabo el proceso de rebase de una rama a la otra. Estas estructuras las denominamos commits_todo y commits_done.
Para poder explicar el procedimiento de cómo se procedió a hacer el comando, ejemplificamos con la siguiente ilustración:

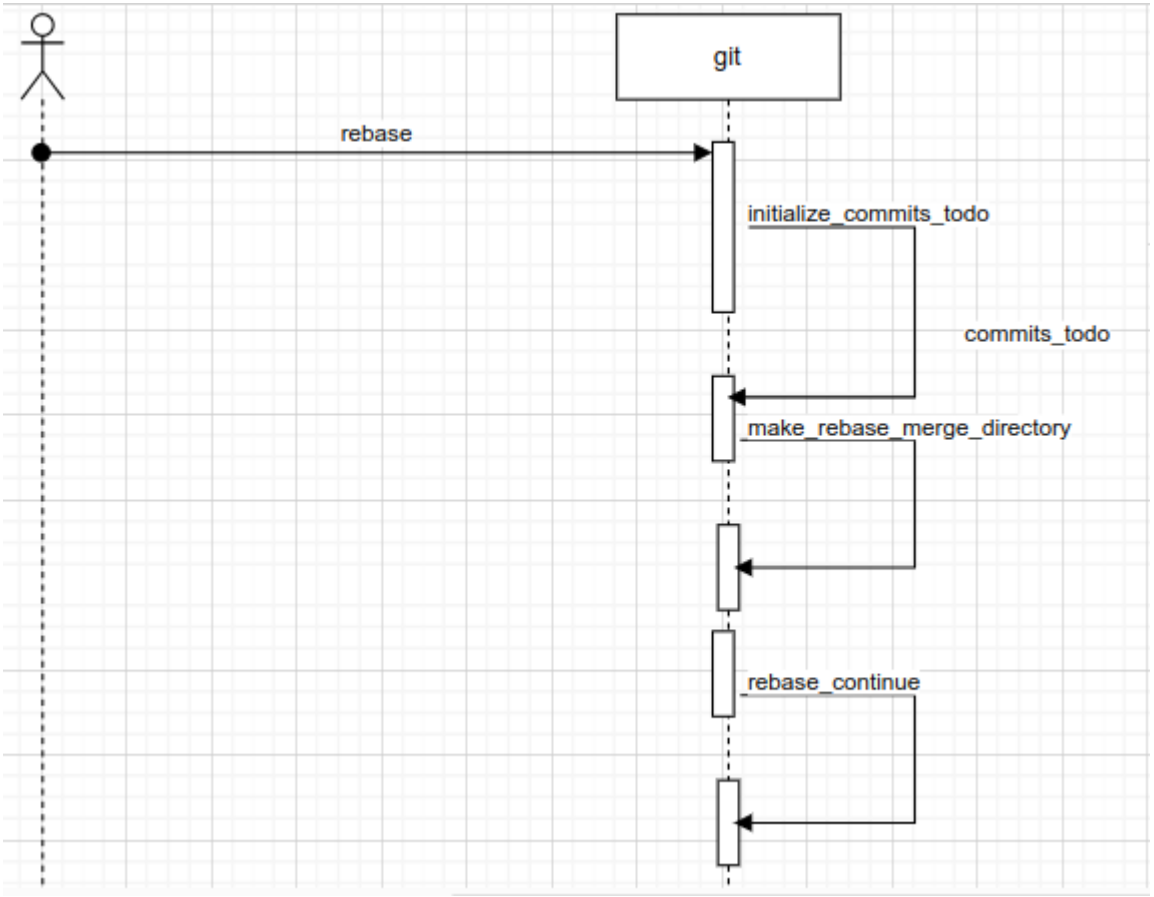


Las funciones principales que utiliza son:

- initialize_commits_todo: Devuelve un HashMap con todos los commits que se encuentran en la rama Main hasta el ancestro común. Para poder conseguirlos, utiliza la función anteriormente mencionada rebuilds_commit_tree.

- make_rebase_merge_directory: actualiza los archivos utilizados para poder dejar registro del estado del rebase en caso de que haya conflictos. Estos archivos van a estar almacenados en la carpeta **rebase-merge**:
  - REBASE_HEAD = guarda commit actual por el que se tiene que empezar a hacer
  - message = guarda el mensaje/nombre del commit que se va a publicar
  - head-name = guarda la rama 'master' (to_branch)
  - git-rebase-todo = guarda los commits en orden temporal ascendente
  - del más príximo al más lejano)
  - author-Script = GIT_AUTHOR_NAME
  - GIT_AUTHOR_EMAIL
  - GIT_AUTHOR_DATE (la hora del commit de from)
  - done = archivo que contiene aquellos commits que ya fueron rebasados

- rebase_continue: Esta función comprueba si el área de preparación no presenta conflictos. Si no hay conflictos, realiza el merge entre el primer commit completo y el último commit realizado, considerando como ancestro al último ‘commit done’ hecho. En caso de conflictos, detiene la ejecución para que el usuario los resuelva y luego ejecute 'rebase –continue'; de lo contrario, continúa con el siguiente ‘commit todo’. Cuando ya no haya más commits completos, finaliza su ejecución.

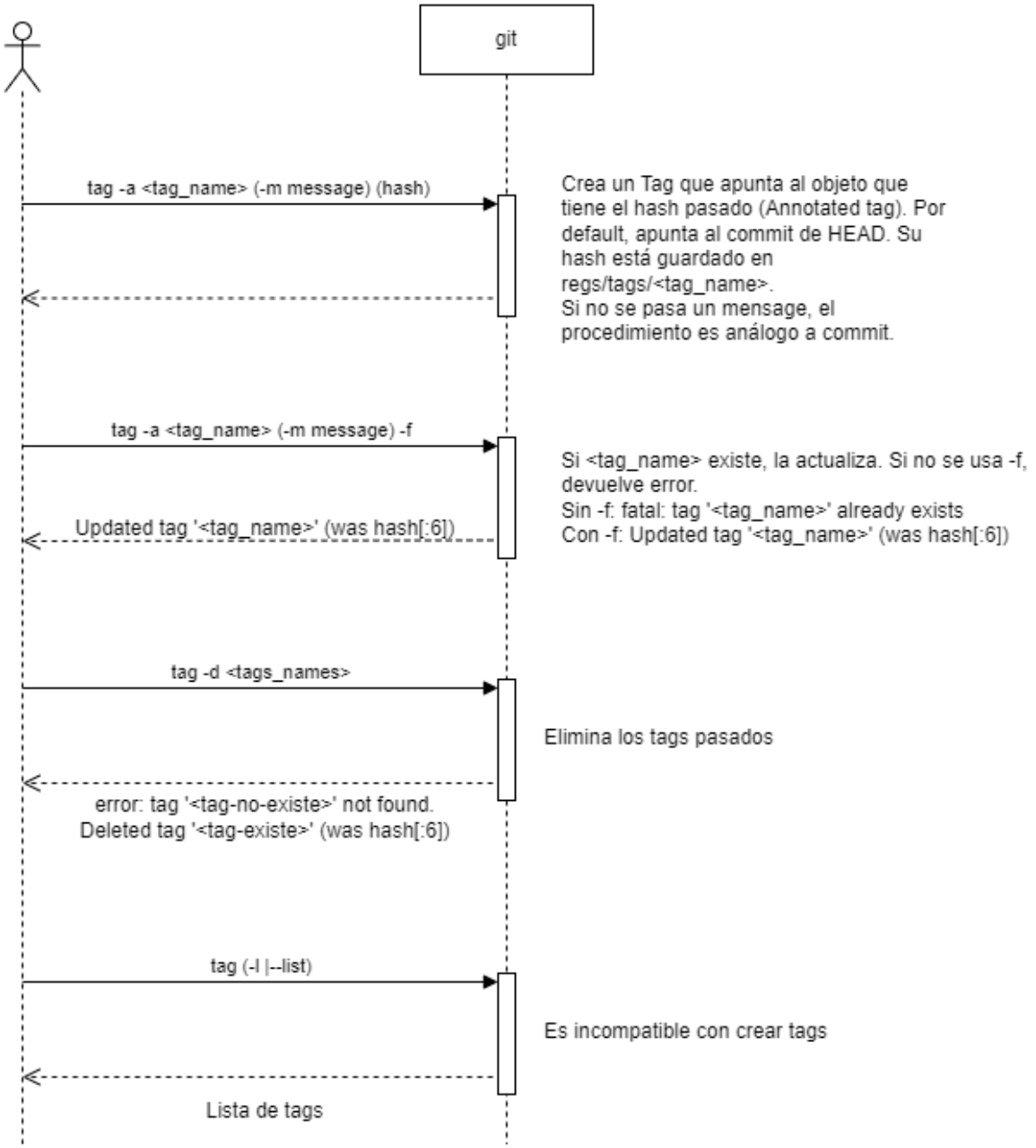


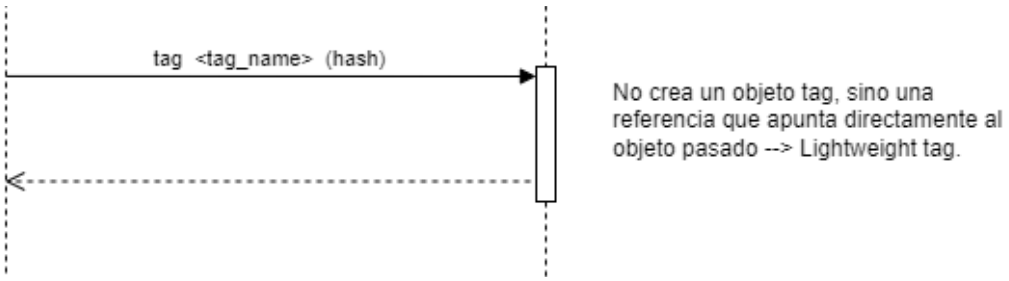
Tag

Existen dos tipos de tags:

- Annotated tags → .git/refs/tags/tag_name apunta al tag creado. Este objeto tiene información relevante, como el tagger, un mensaje, timestamp, offset, y la referencia a la que apunta. Generalmente los tags referencian un commit, pero también admiten otros tipos de objetos git.
- Lightweight tags → A diferencia de las anteriores, no se guardan en la base de datos y .git/refs/tags/tag_name apunta directamente al hash del objeto al que referencia el tag.

Nuestra implementación admite la creación de ambos tipos de tags, así como listar todas las existentes o eliminar algunas.





Conclusión

A través de la implementación de comandos esenciales, se ha logrado proporcionar al usuario final la capacidad de crear, clonar, actualizar y enviar actualizaciones a repositorios remotos. La incorporación de una interfaz gráfica GTK ha añadido una capa visual intuitiva para realizar estas operaciones, brindando a los usuarios una experiencia más amigable y accesible para administrar cambios y comprender la historia de los commits.