Taller de programación

(75.42) Curso Deymonnaz

Trabajo Práctico
Gitr

4 de diciembre de 2023

Gianni Boccazzi
109304

Juan Manuel Dalmau Rennella
109555

Valentín Schneider
107964

Bruno Starnone
108018

Índice

1. Introducción

El presente informe tiene la intención de describir la estructura utilizada para resolver la consigna propuesta por el equipo docente de Taller de Programación 1 - Cátedra Deymonnaz, la cual indicaba la creación de un sistema gestor de versiones similares a Git, el más famoso comercialmente. Se pidió que se desarrolle en lenguaje de programación Rust, por lo que bautizamos a nuestro proyecto como GitR (notación utilizada ampliamente a lo largo del proyecto).

Vamos a describir la estructura del proyecto, detallando los componentes principales de la arquitectura, sus responsabilidades y qué partes del proyecto solucionan.

Luego se hará un repaso por los módulos desarrollados y los comandos pedidos para dar detalle del tren de pensamiento que impulsó cada implementación.

Se entiende que el lector está al tanto de la consigna y las cosas que se solicitaron por lo que no se ahondará en detalle sobre eso en el informe.

2. Estructura

La estructura del proyecto sigue lo que consideramos una arquitectura básica cliente-servidor.

Ambos componentes (cliente y servidor) se comunican a través del protocolo git-transport, como fue solicitado.

Luego implementamos una interfaz gráfica utilizando GTK-3 para Rust y Glade.

Una de las principales banderas de nuestro proyecto es su interoperabilidad. Varias decisiones (y algunos contratiempos) surgen de perseguir esta característica para que lo que nosotros hagamos en nuestros repositorios pueda ser manipulable por el cliente original de git, logrando que nuestras pruebas sean muy robustas al momento de replicar escenarios tanto en git real como en nuestra implementación, verificando que se llega a los mismos resultados.

2.1. Cliente

La parte del cliente lleva la responsabilidad de manejar los repositorios para cada cliente.

Como git es un sistema basado en archivos, gran parte de la implementación pasa por crear, leer, modificar y eliminar archivos; y así mantener una historia de commits, generar un flujo de trabajo con branches y permitir los merges.

El cliente se compone por módulos que manejan estos archivos, facilitando la escritura de los mismos mediante estructuras que representan los objetos fundamentales de git.

El cliente maneja toda la parte de, por ejemplo, crear un blob, un tree o un commit, comprimir sus datos y generar los hashes correspondientes. Luego se encarga de guardarlos en sus correspondientes carpetas.

Todos los comandos implementados pasan por el cliente, que los toma de la interfaz gráfica o la línea de comandos e interactua con el usuario de ser necesario.

En resumen, el cliente es una parte del programa que puede obrar de forma independiente al servidor y puede ser utilizado sin ningún problema para crear repositorios y manejarlos.

2.2. Server

En este proyecto, el servidor cumple un papel totalmente pasivo. Nuestro servidor Gitr se conecta a un Socket local en el ordenador, específicamente en localhost:9418, y permanece a la espera de conexiones por parte de los Clientes Gitr. Al establecer la comunicación, se implementa un protocolo estandarizado compartido entre los servidores Git, conocido como Git Transport. Este protocolo impone diversas reglas que deben cumplirse, como el formato preciso de los mensajes y las distintas etapas o pasos que deben llevarse a cabo durante la interacción.

El propósito fundamental del servidor es gestionar las solicitudes entrantes de los clientes y proporcionar un entorno donde la transferencia de datos pueda tener lugar de manera eficiente y confiable. A través del Socket

local, se establece una conexión bidireccional que permite la transmisión de información según las especificaciones del protocolo Git Transport.

Es importante destacar que el protocolo no solo define el formato de los mensajes, sino también la secuencia de acciones que deben realizarse para garantizar una comunicación efectiva. Esto incluye la autenticación, la verificación de integridad de los datos y otros aspectos críticos para el correcto funcionamiento del intercambio de información entre el servidor Git y sus clientes.

2.3. Interfaz gráfica

La interfaz gráfica tiene la intención de cubrir dos cuestiones principales: manipulación de branches (checkout y visualización de commits) y luego poder resolver conflicts generados por la distintas operaciones (edición de archivos y guardado de los mismos).

Además, queremos que cubra todos los casos de uso básicos y frecuentes de Git como inicializar un repositorio nuevo, commitear cambios e interactuar con un servidor.

A modo de inspiración utilizamos Github Desktop, con sus botones en la parte superior. Nuestra interfaz soporta varias características interesantes:

- Verificaciones de login, si no está el archivo de configuración correcto, genera un warning y solicita que se complete.
- Puede retomar un repositorio ya hecho, replicando el hecho de abrir el programa luego de haber trabajado, tomando el repositorio actual y cargando lo necesario para seguir con el trabajo hecho sin problema.
- Al momento de realizar un merge, si hay conflicts, es informado por un cuadro de diálogo y la interfaz tiene soporte para arreglarlo.
- Las branches se cargan luego de elegir un repositorio y se puede acceder a su historia de commits para tener una presentación más amigable de los mismos

Luego de varias iteraciones y uso encontramos la interfaz muy útil para la creación de commits e historias divergentes que replican escenarios que queremos probar para generar conflicts y casos específicos.

2.4. Protocolo

Como se mencionó anteriormente, el cliente y el servidor tienen la necesidad de comunicarse para poder intercambiar información y lograr la buscada centralización de los repositorios remotos en el servidor, para poder ser accedidos con distintos clientes.

La manera que tienen estos componentes de comunicarse es a través de sockets TCP, utilizando el protocolo propio de git, git-transport.

Este protocolo define varios aspectos de la comunicación que presentaron desafíos diversos:

- Siempre se intenta mandar el mínimo de información posible. Claramente una solución simple sería siempre enviar todos los archivos de un repositorio de un lado a otro, y listo. Pero git-transport limita este aspecto para minimizar el tamaño de lo enviado.
- La conexión se establece con dos comandos, enviados desde el cliente al servidor: git-upload-pack para enviar al cliente y git-receive-pack para recibir desde el cliente.
- Los comandos y mensajes enviados, deben cumplir con el formato pkt-line, que establece una cierta forma para enviar los mensajes.
- Una vez establecida la conexión, se genera un proceso llamado ref-discovery en el cual el cliente y el servidor intercambian información para determinar el mínimo número de cosas a enviar y recibir.
- Cuando se establece lo que se tiene que enviar y recibir, se pasa a la parte de envío de paquetes, cuya estructura consiste en un header, los objetos enviados y un checksum. Los detalles de la estructura ya fueron discutidos a lo largo del cuatrimestre pero se puede revisar la documentacion de git pack-file para conocer los detalles.

Ya presentadas las características, la implementación logró cumplir con todo y comunicar satisfactoriamente un servidor git-daemon (un servidor de pruebas ejecutado por git, que se comunica a través del protocolo) con nuestra implementación de cliente.

Lo implementado para esta parte cumple con la responsabilidad de abrir un socket, establecer la comunicación con el servidor, codificar el mensaje inicial y luego recibir las referencias, decodearlas para que sea más fácil para la parte del cliente manipularlo, y luego lo mismo con el pack-file. Se reciben las cosas por el socket y se desarman para conseguir estructuras más fáciles de manipular desde el cliente.

Los detalles de la implementación se repasan en la sección de módulos, pero a grandes rasgos nos guiamos recibiendo el pack-file del daemon, desmenuzarlo con distintas herramientas y pruebas para poder recibirlo y luego entender cómo armar el nuestro para enviarlo y que git-daemon lo entienda.

3. Módulos

3.1. handler.rs

Este modulo es la entrada controlada del usuario a las funcionalidades del cliente. Desde el main, se llama constantemente en loop a este modulo, que recibe el comando y las flags ingresadas por el usuario. Para cada entrada recibida, se encarga de llamar al comando correspondiente.

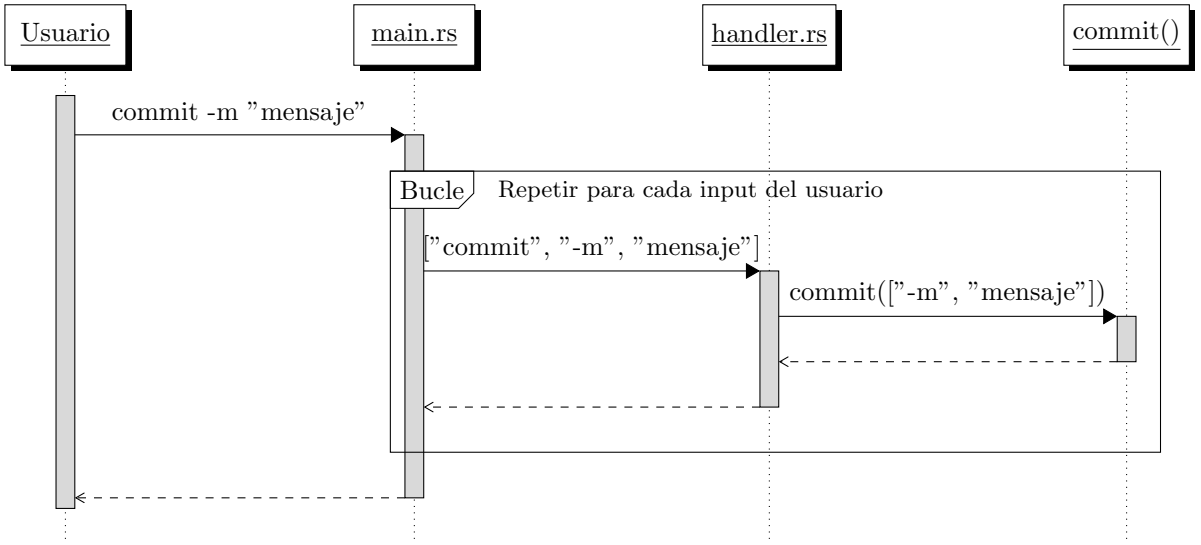

Figura 1: Diagrama de Secuencia para el `command_handler` llamando a `commit`

3.2. commands.rs

Este módulo contiene la implementación de cada uno de los comandos de Gitr. En conjunto con `command_utils.rs`, se desarrolla toda la lógica necesaria para que los comandos funcionen de manera adecuada. Las funciones definidas en este módulo son las que el *handler* llama para gestionar la ejecución de los comandos.

3.3. Git Objects

Los objetos son la célula fundamental de la implementación. Acá es dónde brilla la interoperabilidad, ya que logramos generar los mismos objetos que generaría un cliente de git real. Logrando los mismos hashes para los mismos archivos, replicando escenarios de git con muy buena precisión.

Es sabido que git utiliza complejos sistemas de hashes, compresión y estructuras de archivos que a priori parecen contraintuitivas o imprácticas, y presentaron un desafío al momento de lograr la interoperabilidad.

Gracias al esfuerzo del grupo por lograr lo propuesto con esta característica, pudimos lograr ver a nivel de bytes cómo estaban estructurados los archivos guardados en un repositorio de git y lograr replicarlos en

la implementación nuestra. Esta destacable visualización de bytes se logró con comandos de terminal que no están presentes en el programa, generando pipelines para los objetos que queríamos ver y estaban en directorios externos al proyecto. Esta herramienta y este manera de encarar los problemas resultó sumamente útil para resolver otras cuestiones como el pack-file.

La interoperabilidad fue un trabajo milimétrico ya que un cambio de un byte, alteraba completamente el hash producido por ese objeto y la interoperabilidad ya no era posible.

En próximas secciones se desarrolla al detalle la estructura de cada objeto y cómo es que tienen que estar estructurados para poder lograr lo planteado.

3.4. commands_utils.rs

El siguiente módulo tiene como objetivo mejorar la legibilidad del código. Cada comando tiene funciones específicas asignadas para realizar tareas particulares. Todas las funciones necesarias para cada comando están organizadas en este módulo, lo que facilita la comprensión del código en el archivo `commands.rs`. Se ha logrado una separación clara y etiquetada de cada función asociada a cada comando.

3.5. file_manager.rs

Este módulo tiene como propósito llevar a cabo todas las operaciones relacionadas con la lectura, escritura, creación y eliminación de archivos. Cada comando de Gitr utiliza el *file_manager* en caso de necesitar modificar archivos. Esta estructura permite establecer un orden claro y facilita la reutilización de funciones en múltiples comandos.

3.6. logger.rs

Este modulo se encarga de actualizar el archivo `log.json`. En el codigo del programa, en ciertas funciones importantes (como las del `file_manager.rs`) las funciones `log_error`, `log_action` y `log_file_operation` son llamadas. Consiste de un struct Logger que representa una linea del archivo log. Cada struct o log puede tener un tipo (Error, Action o FileOperation) para una mejor comprension a la hora de leer el `log.json`. Ademas, existe una funcion implementada para poder ver el output del log durante la ejecucion del cliente. Para usarla, se debe llamar `l -<amount-of-logs>`.

Ejemplo del output que se escribe en log al llamar al comando `branch -d master` (resumido, porque hay varias llamadas al archivo `.head_repo`)

```
1 {"type": "Action","timestamp": "03-12-2023 17:52:47","message": "calling branch with flags:
    ["-d", "master"]"}
2 {"type": "FileOperation","timestamp": "03-12-2023 17:52:47","message": "reading data from:
    client/.head_repo"}
3 {"type": "FileOperation","timestamp": "03-12-2023 17:52:47","message": "reading data from:
    client/repo/gitr/HEAD"}
4 {"type": "Error","timestamp": "03-12-2023 17:52:47","message": "ERROR: No se puede borrar
    branch 'master': HEAD apunta ahi"}
```

4. Comandos

A continuación, un listado de los comandos implementados y las decisiones de diseño tomadas para cada uno.

4.1. hash-object

El comando Hash-Object recibe la data un archivo existente en el repositorio y devuelve su hash. Para poder implementarlo, simplemente hasheamos la data con sha1 y la devolvemos. En el caso de que se use la flag -w, ademas el comando escribe el archivo en disco. Y para eso entonces, ademas debemos comprimir la data porque asi es como debe ser guardada. Aca estamos aprovechando los structs de los objects que implementamos, porque al recibir la data del archivo nos creamos un blob, y si hace falta guardar, simplemente llamamos al metodo `blob.save()`.

4.2. init

Este comando se encarga de inicializar todo lo necesario para un repositorio.

Git lleva una estructura específica de directorios para guardar archivos y archivos específicos para llevar las "heads"que son los commits a los que apuntan cada cabeza de branch y llevar tracking de dónde se está trabajando.

Init se encarga de crear todos estos directorios y archivos para dejar un repositorio listo para empezar a usar.

Internamente, llama a varias funciones de `file_manager.rs` para crear los directorios, archivos en ellos y escribirlos.

En caso de no poder crear algún archivo o tener algún tipo de problema, se emiten errores correspondientes propagados hasta la función llamadora.

4.3. cat-file

Este comando es casi la inversa de Hash-Object, recibe un hash y devuelve su data. Puede ser el hash de cualquier object. Segun las flags, puede devolver el tipo (-t), el tamaño (-s), o la data (-p). Para implementarlo simplemente buscamos el archivo a partir del hash recibido, los abrimos, descomprimimos la data y ya la tenemos lista. En el caso del tree es un poco mas complejo, porque los hashes de sus entries estan escritas en binario, asi que para ese caso particular ademas nos tomamos el trabajo de traducir a algo human-friendly.

Por ejemplo, corriendo `cat-file -p <tree-hash>` se obtiene:

```
100644 blob <hash> archivo1
40000 tree <hash> dir
```

Como se puede observar, en la ultima columna se muestra solo el nombre del archivo/carpeta, no se muestra el path completo. Y tambien, si la carpeta tiene mas archivos/carpetas adentro, estas no se ven reflejadas en este tree, para acceder a esa informacion, se deberia volver a llamar a cat-file con el hash de ese tree.

4.4. add

El comando `add` agrega archivos al index y crea los blobs correspondientes. Cada repositorio Gitr contiene su propio index en la carpeta `gitr`. Este archivo consiste en una lista de blobs listos para ser commiteados, con el siguiente formato:

```
100644 bf31ad2a34f2f41ac5b90d86c3f9e151bd043fe2 0 cliente/repo/archivo1
100644 3179e117577e8a686c035d0c2ec0f73605e9a3bc 0 cliente/repo/archivo2
```

Dicho archivo index se encuentra comprimido con la misma función de compresión utilizada en los objects.

Se puede utilizar `add <archivo>` para agregar un archivo al index o `add .` para agregar todos los archivos del repositorio al índice y crear los blobs correspondientes, excluyendo aquellos definidos en el archivo `gitrignore`. En caso de volver a agregar un archivo que ya estaba en el índice, se volverá a incluir si fue modificado.

La función `update_index_before_add()` tiene como finalidad garantizar que el index (*index*) no contenga archivos que no estén presentes en el directorio de trabajo (*working directory*) en el momento de ejecutar el comando `add`.

4.5. rm

El comando `rm <archivo>` tiene la función de eliminar un archivo del index. El proceso se realiza de la siguiente manera:

- Descomprimir el index para acceder a su contenido.
- Verificar si el archivo a borrar se encuentra en el index.
- En caso de encontrar el archivo, se reemplaza la línea correspondiente por una cadena vacía.
- Comprimir nuevamente la información y guardarla en el index.

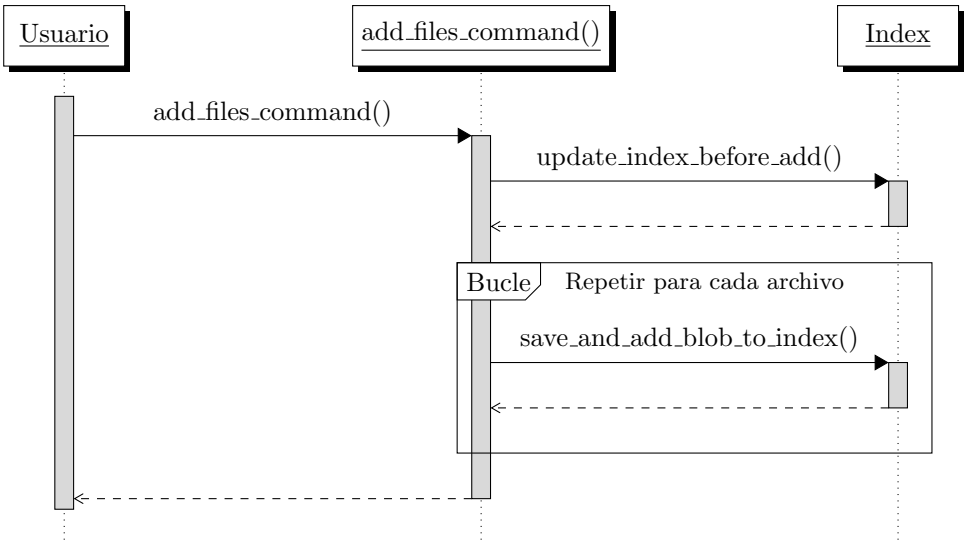

Figura 2: Diagrama de Secuencia para el Comando `add`

Este procedimiento asegura que el index no contenga la referencia al archivo que se desea eliminar, manteniendo la consistencia entre el índice y el estado actual del repositorio.

4.6. commit

La implementación del comando `commit` consta de los siguientes pasos:

- Si no hay un index, se muestra el estado actual (*status*).
- Si hay un index, se verifica utilizando las funciones `get_untracked_notstaged_files()` y `get_tobe_commited_files()` si hay cambios respecto al commit anterior (si existe). En caso afirmativo, se crea con el commit.

Para realizar el commit, se utiliza la función `get_tree_entries`. Esta función realiza los siguientes pasos:

1. Llama a `get_hashmap_for_checkout()`, cuya finalidad es devolver un *hashmap* de todos los blobs presentes en el index. Como el index contiene rutas completas, esta función lee cada ruta y, al ordenar, agrupa los archivos en cada directorio. La clave representa el directorio y el valor es un vector de archivos.
2. Con el *hashmap* obtenido, se llama a `create_trees()`, que, de manera recursiva, crea los objetos *trees* para cada directorio. Primero se crean los árboles de los subdirectorios y luego los de los directorios padres. Finalmente, se devuelve el *main tree*.
3. Utilizando el main tree, se llama a `write_new_commit_and_branch()`, que crea un objeto commit y lo guarda en el directorio `objects`. En caso de ser el primer commit, crea el archivo `master` en `gitr/refs/heads` y le asigna el *hash* del commit creado.

4.7. checkout

La implementación de `checkout` sigue estos pasos:

1. Verifica la existencia de al menos un commit, ya que sin un commit no sería posible crear una rama ni realizar el checkout.
2. Llama a `get_branch_to_checkout()`, que comprueba la existencia de la rama a la que se realizará el `checkout`. En caso de usar la opción `-b`, crea la nueva rama.

3. Utiliza la función `get_commit()` del *file manager*, la cual busca en `refs/heads/` la rama a la que se realizará el `checkout` y devuelve el último commit asociado con esa rama.

4. Con el último commit obtenido, se invoca a `update_working_directory` del *file manager*. Esta función, a su vez, llama a `delete_all_files`, que borra todos los archivos del directorio de trabajo (*working directory*), preparándolo para los archivos asociados al commit de la rama especificada.

5. Comienza a recorrer el *main tree*, creando cada directorio y archivo que contiene utilizando las funciones `create_tree` y `create_blob`. De este modo, se restaura el directorio de trabajo de manera idéntica al commit de la rama especificada.

De esta forma se asegura que el working directory se actualice correctamente según el commit de la rama seleccionada durante el `checkout`.

4.8. log

La finalidad del comando es visualizar el historial de commits de la rama en la que se encuentra el usuario. La implementación de este comando incluye los siguientes pasos:

1. Verificar con la función `commit_existing()` si existe al menos un commit creado en la rama actual.

2. En caso afirmativo, iterar siguiendo estos pasos:

   a) Leer el commit.

   b) Imprimir la metadata con sus respectivas funciones.

   c) Leer el hash del padre del commit leído.

   d) Repetir estos pasos hasta que se alcance un commit que no tiene padre, indicando que se ha llegado al commit root, o alcanzando el límite pedido de historial.

Este proceso permite al usuario explorar de manera secuencial los commits en la rama actual, comenzando desde el commit más reciente hasta llegar al commit raíz.

4.9. status

El comando status informa las diferencias que hay entre archivos en el working directory actual, en el index (o staging area) y en el ultimo commit realizado. Estas diferencias son categorizadas en `to be commited`, `unstaged` y `untracked files`. Para cada categoria se comparan los 3 conjuntos de archivos como corresponda. Para implementarlo, creamos HashMaps a partir de los 3 conjutnos de archivos (clave: path, valor: hash), e iteramos por cada uno preguntando si el archivo existe o no en algun otro, o si existe pero su hash es distinto, etc.

4.10. clone

La función `clone` se configura con dos parámetros: el nombre del repositorio que se va a clonar y el nombre del nuevo repositorio local que se desea crear. Su función principal consiste en inicializar un nuevo repositorio Gitr con el nombre proporcionado. Posteriormente, se establece el repositorio proporcionado como remoto, y se inicia un proceso de "git-upload-packçon el servidor correspondiente para solicitar la transferencia de todos sus objetos asociados. Este proceso garantiza la sincronización completa de los datos entre el repositorio remoto y el nuevo repositorio local, estableciendo así una réplica fiel del repositorio original.

4.11. fetch

El comando `fetch` realiza una acción similar a un `pull`. Al ejecutar este comando, el cliente establece comunicación con el servidor y solicita la transferencia de todos los objetos asociados que no se encuentran aún en el repositorio local. A diferencia de un `Pull`, el `Fetch` no actualiza directamente el Directorio de Trabajo; sin embargo, permite tener disponible localmente todas las ramas de trabajo presentes en el servidor remoto. Este

proceso es valioso para mantener actualizada la información del repositorio local sin modificar directamente los archivos de trabajo. *(podria agregar que pull y fetch llaman practicamente a lo mismo y que se manejó esa diferencia con una flag)

4.12. merge

El comando Merge recibe una rama existente y junta los contenidos de la rama actual con la recibida. (Nota: en esta seccion llamaremos Master a la rama en la cual estamos parados y New a la rama con la que se desea hacer el merge) Dependiendo de la historia de los commits, automaticamente se ejecutaran diferentes tipos de merge segun corresponda. Para esto, recorremos todos los commits de New de mas reciente a menos reciente y preguntamos si pertenece a Master. El primer commit que pertenezca a ambos nos servira para darnos cuenta en que caso estamos, y si estamos en three-way merge, sera el commit que utilizaremos como base.

4.12.1. Fast-Forward merge

Si la historia de los commits es como se muestra en la imagen, donde se crea una branch pero no hubo nuevos commits en la rama principal:

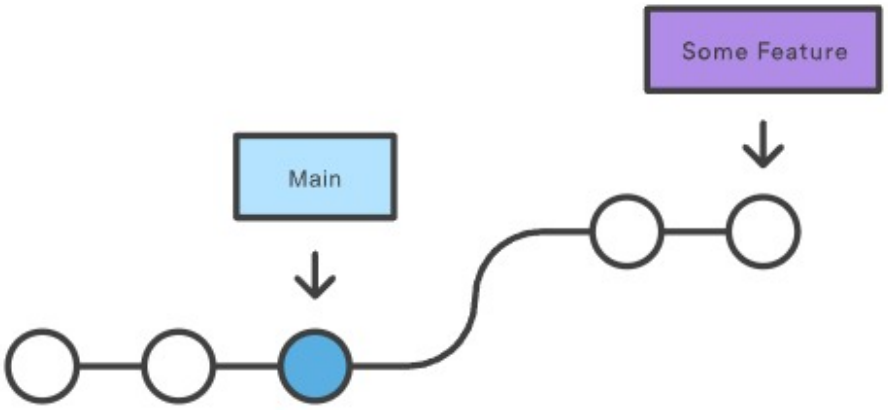

Figura 3: Caso Fast-Forward Merge

Entonces se ejecuta fast-forward merge, que simplemente consiste en mover el tip de Master al tip de New. Y luego actualizamos el working directory como corresponde, para que refleje el ultimo commit.

4.12.2. Three-Way merge

Si en cambio, en Master hubo nuevos commits luego de crear la rama:

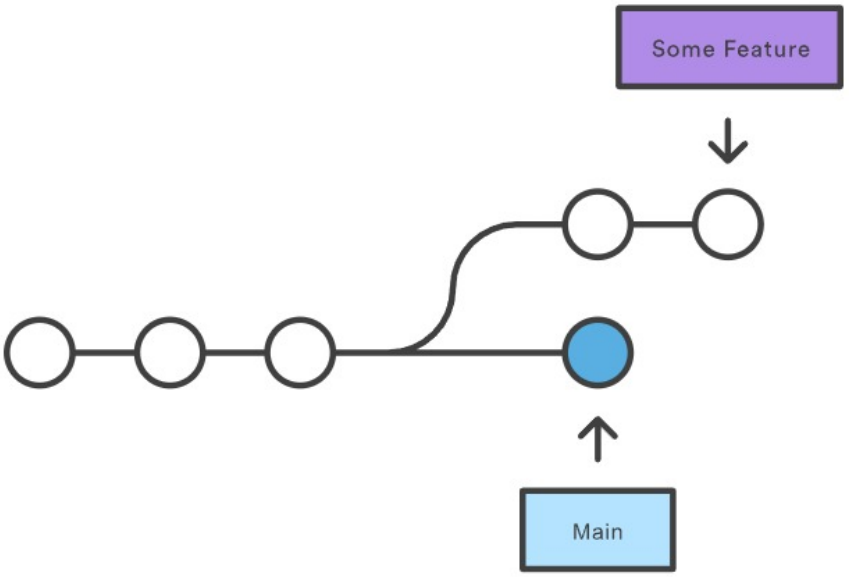

Figura 4: Caso Three-Way Merge

Estamos en el caso de Three-Way Merge. Este caso es mas complejo, porque involucra realmente juntar ambos commits. Primero que nada, si hay archivos en New que no existen en Master, son agregados. Luego iteramos por los archivos de Master y los comparamos con los de New. Para cada archivo, si los hashes coinciden, es porque no hubo cambios, asi que continuamos. Ahora, si los hashes no coinciden, es donde tomamos el commit base. Para eso, utilizamos la siguiente logica.

MERGE BASE	A	B	3-WAY MERGE
X	X	B	B
X	A	X	A
X	X	X	X
X	Y	Y	Y (A/B)

Figura 5: Logica para decidir con que archivo quedarse

Suponiendo que A es Master y B es New en nuestro ejemplo. Es decir, si New cambio un archivo que Master no, o al reves, siempre me voy a querer quedar con el cambio. Si ninguno cambio o ambos cambiaron exactamente lo mismo, me quedo con cualquiera, total es lo mismo.

Pero, ¿como me "quedo con el cambio"?. Con Diffs. Un Diff es una receta para modificar un archivo. Por ejemplo, un diff entre base y Master nos diria que lineas agregarle y/o eliminarle a base para que quede como Master.

En nuestra implementacion, un Diff es un struct, al crearse, se consiguen las diferencias que se deben aplicar al archivo en un vector.

Para poder conseguir el diff de un archivo, implementamos un algoritmo llamado **"largest common subsequence"**. Luego, la funcion `aplicar_diffs()` recibe un archivo y un Diff, entonces aplica el Diff a ese archivo y lo guarda en disco.

Ahora, queda un ultimo caso mas para cubrir, que es el siguiente:

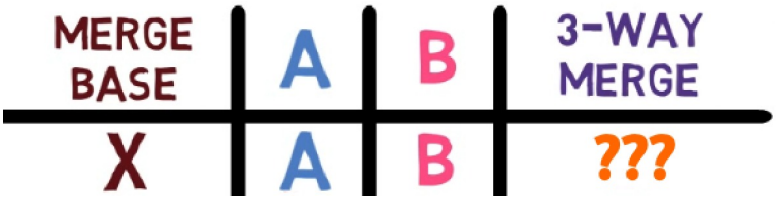

Figura 6: ¿Que pasa si ambos cambiaron el mismo archivo?

En este caso tenemos un Conflict. Asi que debemos decidir que hacer.

Lo que hacemos es crear un Diff entre Base-Master y Base-New. Para facilitar la implementacion y aprovechar las funciones ya hechas, lo que hacemos es juntar estos diffs en uno solo, y volver a llamar a `aplicar_diffs()`. Para juntarlos, utilizamos `comparar_diffs()`. Esta funcion compara solamente los cambios que ambas ramas hicieron sobre los archivos, por lo que es mas facil ver en que momento generar el conflict: en el momento en el que ambos diffs intenten agregar cosas distintas sobre la misma linea.

Bueno, no es tan simple en realidad, eso es solo el comienzo de un conflict. Si la siguiente linea de ambos diffs es consecutiva a la anterior, y hay un conflict abierto, entonces esas lineas tambien son parte. Y asi hasta que las lineas dejen de ser consecutivas, independiente de cada branch. Es decir, si tenemos los diffs:

```
BASE-MASTER DIFF
1.+edicion
2.+en
3.+master

BASE-NEW DIFF
1.+edicion en new, aca lo hago solo en una linea
```

El conflict resultante se hara entre esas 3 lineas de Master contra esa unica de New. Y finalmente, para poder reutilizar `aplicar_diffs()` necesitamos mantener la estructura de un Diff comun, por lo que cuando tenemos un conflict lo juntamos todo en una unica linea.

```
UNION DIFFS
1.+<<<<<<< HEAD\nedicion\nen[...]=======\nedicion en new...
```

Y luego, este diff nuevo lo recibe `aplicar_diffs()` que se ejecuta normalmente.

4.13. remote

El comando `Remote` ofrece una flexibilidad esencial al permitir gestionar y determinar el repositorio remoto con el cual el cliente desea comunicarse durante solicitudes y envíos de datos. Cuando se utiliza sin argumentos adicionales, este comando imprime en pantalla el repositorio remoto actualmente configurado para la comunicación.

Esta capacidad de elección proporciona un control significativo sobre las interacciones del cliente con repositorios remotos, permitiendo ajustar dinámicamente las conexiones según las necesidades del proyecto. Además, la funcionalidad de impresión sin argumentos facilita una rápida visualización del repositorio remoto actual, brindando claridad sobre la configuración de comunicación en un momento dado.

4.14. pull

El comando `Pull` desempeña un papel crucial en el desarrollo de este proyecto, destacándose como uno de los comandos principales debido a su funcionalidad esencial: la solicitud de objetos al servidor mediante la ejecución del protocolo Git Transport. Esta operación implica la comparación de los commits actuales en el cliente con los del servidor, determinando la necesidad de actualizar sus objetos.

Cuando el repositorio local se encuentra desactualizado en relación con el servidor, el comando `Pull` inicia una solicitud para obtener el packfile mínimo necesario.

La ejecución precisa del protocolo durante un `Pull` asegura la coherencia entre el repositorio local y el servidor remoto, manteniendo el historial de commits sincronizado. Este proceso se vuelve fundamental en situaciones donde múltiples colaboradores contribuyen al proyecto, garantizando que cada cliente obtenga las actualizaciones más recientes de manera eficiente.

4.15. push

El comando `push` se destaca como una herramienta esencial en el proyecto, permitiendo al cliente enviar nuevos objetos al servidor para que estén disponibles para otros colaboradores. Esta funcionalidad es clave para la colaboración efectiva, ya que posibilita la propagación de cambios entre diferentes instancias del repositorio.

Cuando se ejecuta el comando `Push`, se inicia un proceso que examina las posibles incongruencias entre los commits actuales del servidor y los del cliente. Posteriormente, determina el packfile mínimo necesario para sincronizar el servidor y el cliente. Este packfile contiene los objetos requeridos de manera eficiente, facilitando una transferencia de datos optimizada.

4.16. branch

Branch muestra, crea o elimina ramas en el repositorio. Para el caso mas simple, listar las branches actuales, el comando itera por la carpeta refs/heads y muestra el listado disponible. Ademas la branch apuntada por HEAD se destaca para mejor entendimiento. Si al comando se le pasa el nombre de una rama inexistente: `branch <new-branch-name>`, se creara una nueva rama con el nombre recibido (a menos que ya exista). Si al comando se le pasa la flag de mover (-m) y 2 ramas, una existente y una que no existe, practicamente lo que termina ocurriendo es que se le cambia el nombre a la branch origen. Finalmente, si se quiere eliminar una rama (-d), el comando busca el archivo en refs/heads y lo elimina (siempre y cuando HEAD no este apuntando a la misma, es decir, que no estemos parados en la branch que queremos eliminar).

4.17. ls-files

La implementación de `ls-files` se basa en tres flags:

1. `--stage`: Esta opción lee el index y devuelve información detallada de todos los archivos presentes en el index en ese momento, incluyendo el modo, el hash, los permisos y la ruta. En este caso, la implementación es tan sencilla como llamar a la función `read_index()` del *file_manager* y devolver su contenido.
2. `--modified / --deleted`: Esta opción muestra los archivos que han sido modificados o borrados desde la última vez que se ejecutó el comando `add`. Para esto, se llama a la función `get_ls_files_deleted_modified()`, la cual, dependiendo del flag proporcionado, devuelve los archivos que han sido modificados o los que han sido eliminados del directorio de trabajo (*working directory*).

4.18. show-ref

El comando `show-ref` proporciona al cliente la capacidad de visualizar en tiempo real a qué commit o tag apuntan sus referencias actuales. Esta funcionalidad es invaluable para obtener una comprensión inmediata de la situación del repositorio y la relación entre los commits y las etiquetas.

Si se utiliza el modificador `--head`, el comando también imprimirá en pantalla el commit al que apunta el Directorio de Trabajo en ese momento. Esta característica adicional facilita una visión más completa de la estructura del repositorio y la posición específica en el historial de commits.

4.19. tag

La implementación de `tag` sigue la misma estructura de los objetos en Gitr. Se utiliza un struct `Tag`, el cual contiene la metadata del tag y además tiene una referencia al commit al cual se le está asignando el tag.

Esta implementación incluye dos tipos de tags:

1. **Lightweight tag**: Este tipo de tag no crea un objeto Gitr, sino que simplemente crea un archivo en `refs/tags` que contiene el hash del commit que está siendo etiquetado. Para crear este tag, se utiliza el comando `git tag <nombre-del-tag>` estando en la rama del commit que se desea etiquetar.
2. **Annotated tag**: A diferencia del tag ligero, este tag es un objeto Gitr. Al llamar al comando `git tag -a <nombre-del-tag>-m <mensaje-del-tag>`, se crea un archivo en `refs/tags` que contiene el hash del tag en lugar del hash del commit. Este archivo incluye el mensaje asociado al tag.

   Además, se pueden usar las flags `-d` para eliminar un tag, y también `tag` para ver la lista de tags.

4.20. ls-tree

El comando ls-tree recibe el hash de un commit y muestra sus contenidos. Dependiendo de ciertas flags que reciba (-r, -t, -z por ejemplo) la informacion se muestra de una u otra manera. Tambien pueden combinarse las flags. Por ejemplo -r llama recursivamente a todos los archivos, y si ademas usamos -t ademas se incluiran los trees. Para combinarlas se debe hacer, por ejemplo: `ls-tree -rt <tree-hash>`. Para lograr este comando, en su forma basica sin flags simplemente se llama a cat-file porque es el mismo comportamiento. Para la implementacion con flags, se obtiene la data y se van agregando o no, segun corresponda lo que las flags digan que se necesita.

4.21. check-ignore

Este comando tiene la función de revisar el archivo `gitrignore` y reconocer si el `path` que le fue pasado por parámetro está dentro de la lista o no.

En el archivo `gitrignore` los paths deben guardarse relativos al repositorio, con / adelante. Por ejemplo, queremos ignorar los cambios hechos en el archivo repositorio/file1, entonces en gitrignore debería aparecer una línea + `/file1` y al utilizar el comando `check-ignore /file1` debería aparecer un mensaje de que ese path está ignorado. Si en cambio ejecutamos el comando con `/file2` como argumento, no debería emitirse ningún mensaje ya que no está ignorado (Este comportamiento es similar al de git real). * los paths que se guardan aca van relativos al repo con / adelante. Ej si el archivo esta en cliente/repo/archivo en el ignore va /archivo.

4.22. rebase

Rebase es un comando que se encarga de unir los cambios en dos ramas distintas pero sin tener un commit con dos padres como lo hace merge. La intención fundamental de rebase es mantener una sola historia lineal para esa unión. Esto se logra de la siguiente manera: si se tienen dos ramas que divergen en un commit A, llamemoslas main y topic. Sobre main se generan dos commits más B y C, y sobre topic otros dos D y E.

Para este caso, debemos pararnos sobre topic y llamar al comando `rebase main` para poder cambiar la base de la historia de topic, a la punta de main.

Rebase logra hacer esto, revisando los commits de topic y aplicando esos cambios a la branch main, a partir del commit C; generando finalmente una historia lineal del estilo: (to-do! agregar grafico del ejemplo este).

En nuestra implementación, esto lo logramos rescatando los commits de topic en un vector y reutilizando las funciones de three-way merge tomando como base el commit A y luego realizando los merge entre los commits que correspondan. Esto se hace así con el objetivo de mostrar conflicts cuando sucedan y manejarlos, ya que pueden ocurrir.

Si esto se hace en git real, al generarse un conflict, el programa pide que lo solucionemos y ejecutemos el comando con la opción –continue para poder seguir con el rebase.

En nuestro caso, se interrumpe la ejecución y se entra en un diálogo con el programa para enviar estos comandos de –continue luego de solucionar los conflicts.