Taller de Programación I - Cátedra Deymonnaz

Segundo Cuatrimestre 2023

Proyecto: Git Rústico

INTEGRANTES
- Juan Pablo Carosi Warburg 109386
- Mateo Daniel Vroonland 109635
- Siro Fatala 109669
- Juan Ignacio Pérez Di Chiazza 109887

Índice

Introducción **1**
Arquitectura **2**
Ejecución Comandos 2
Estructura de cada comando 2
Flujo de comunicación cliente/servidor 3
Estructuras y sus funcionamientos: 3
Estructura de objetos 4
Detalles de la implementación 5
Merge 5
Rebase: 6
Write-tree: 7
Logger 8
Aprendizajes y conclusiones **9**

Introducción

Como es sabido, este cuatrimestre tuvimos que implementar una versión de git más rústico en Rust. Durante este informe vamos a estar explicando cómo implementamos las funcionalidades pedidas y las decisiones que tomamos con respecto a nuestra implementación.

Arquitectura

Ejecución Comandos

Para modelar los comandos decidimos hacer un enum que contiene los mismos, y luego, llamamos a ejecutar el comando correspondiente según la ocasión. Otra alternativa interesante es realizarlo de una manera polimórfica, pero, luego de realizar un análisis más en profundidad, nos decantamos por la primera opción ya que esta resulta ser más performante.

Sin embargo, decidimos implementar también este enfoque polimórfico mediante la implementación de traits ya que con esto hemos asegurado que cada comando tenga una interfaz/firma consistente, lo que facilita su ejecución en situaciones diversas sin tener que preocuparnos por los detalles específicos de cada comando.

Al utilizar un enum, podemos aprovechar la eficiencia y la simplicidad de esta estructura de datos para representar de manera clara y concisa cada tipo de comando. Esto se debe a que las enumeraciones en Rust suelen ser más eficientes en términos de uso de memoria y velocidad de ejecución en comparación con un enfoque más orientado a objetos.

Estructura de cada comando

Para modelar cada comando nos decantamos por una opción bastante simple pero eficaz, un struct por comando. Este está preparado para contener campos internos ya sea estructuras que van a facilitar el funcionamiento de nuestro comando, indicadores de flags presentes para nuestro comando o parámetros pedidos.

Un ejemplo puede ser el siguiente:

```
pub struct add
    logger: Arc<Logger>,
    ubicaciones: Vec<PathBuf>,
    index: Vec<ObjetoIndex>
```

En este caso las ubicaciones serían archivos/directorios pasados por parámetros y por otro lado, el index sería una estructura auxiliar que nos va a facilitar la ejecución del comando. Esto se debe a que nos permite acceder fácilmente a los objetos que están dentro del index evitando tener que leerlo múltiples veces.

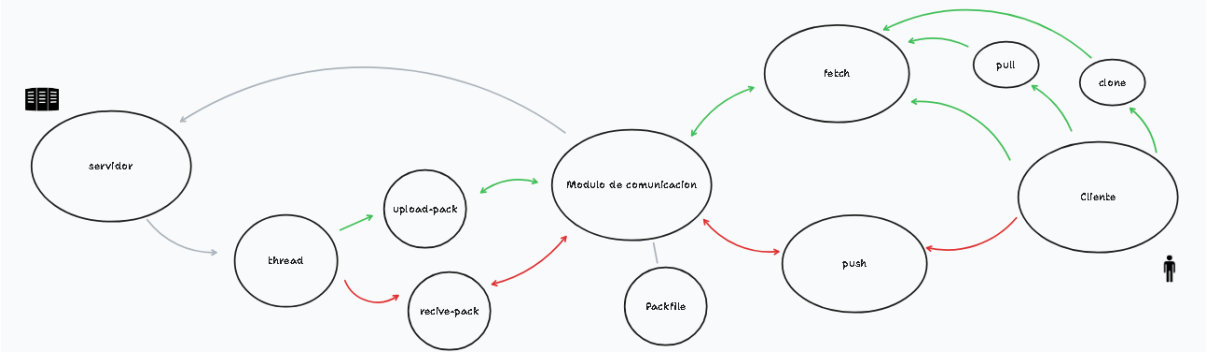
Tomamos este ejemplo para también explicar brevemente la estructura del index y cómo lo decidimos modelar ya que es muy diferente comparado a lo que propone y utiliza git. Determinamos usar otro modelo ya que notamos que el index de git tiene demasiada información que no íbamos a utilizar y, por lo tanto, complejizaba las tareas que involucran el index sin añadir beneficios.

[tipo] [merge] [modo] [hash] [ruta_archivo]

Con la imagen de arriba se puede observar cómo se modela cada archivo que entra a la zona de staging, o sea, el index. Cada campo tiene un propósito particular, los cuales son:

- tipo = puede tomar los valores + o -, indica si es una addition o una deletion
- merge = puede tomar los valores 1 o 0, indica si el archivo tiene conflictos o no
- modo = indica el tipo de archivo añadido, por ejemplo un modo válido puede ser 100644
- hash = un hash de 40 dígitos en formato hexadecimal que sirve para acceder al objeto asociado de esa ruta
- ruta_archivo = indica el path del archivo añadido

Flujo de comunicación cliente/servidor



Con este diagrama se pretende dejar en claro como es el funcionamiento y la interacción entre las partes para llevar a cabo la comunicación entre el servidor y el cliente, donde se puede observar que ambas utilizan el módulo comunicación como API.

Estructuras y sus funcionamientos:

- **Struct Servidor**: se encarga de procesar las peticiones haciendo uso de las funciones upload-pack y receive-pack, y de enviar errores en caso de que sea necesario.

- **Struct Comunicación**: como se mencionó anteriormente, cumple el rol de intermediario entre cliente-servidor, facilitando la comunicación entre partes, se optó por usar los primeros 4 bytes para determinar el largo de la línea, y no leer líneas para no depender de si se manda o no el /n, evitando futuros problemas.

- **Cliente**: es el que ejecutará los comandos, los cuales iniciarán la comunicación con el servidor y se encargarán de cumplir con su propósito tal y como funcionan en la aplicación comercial.

- **Struct Packfile**: estructura que se utiliza para enviar y recibir objetos, se cuenta con la lectura de delta-refs pero no su construcción.

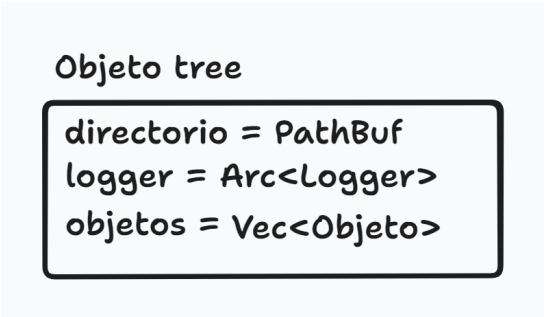
Estructura de objetos

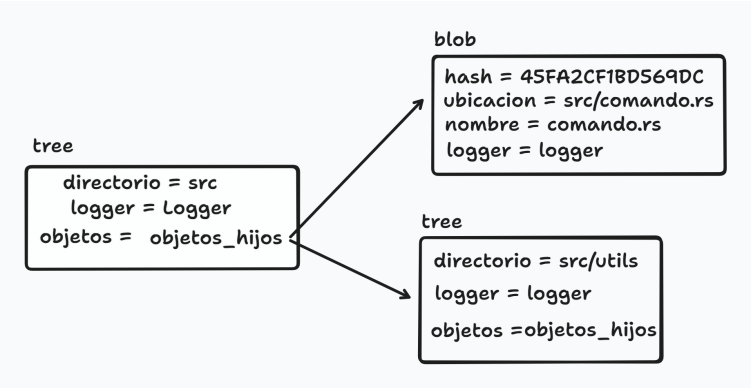
Consideramos pertinente explicar esta idea en una sección aparte ya que es de las ideas más importantes del proyecto. Si bien es una abstracción que se nota fácilmente, nos fue de gran ayuda a la hora de realizar el proyecto.

Durante la semana de investigación notamos que había cuatro posibles objetos que puede tener un repositorio de git, siendo ellos: blob, tree, commit y tag. Luego, a medida que fuimos implementando comandos, vimos que la ejecución del mismo, en la gran mayoría de casos, debe cambiar según el tipo de objeto presente, es por esto que encapsulamos estos diferentes comportamientos dentro de cada tipo de objeto.

Además, estas primitivas para cada tipo de objeto fueron de utilidad para reciclar patrones que se repiten en diversos comandos, como por ejemplo, si un tree contiene determinado hijo.

Cabe destacar que si bien son todos objetos, manejan distintas complejidades, es por esto que nos vamos a explayar un poco más con los objetos trees y commits. Por el lado del tree, este tiene una estructura bastante simple pero, tiene una gran cantidad de primitivas.





En las imágenes de arriba se puede observar cómo es la estructura de un árbol y como se comporta al tener otros objetos dentro.Cabe recalcar que la gran mayoría de las primitivas de los árboles las realizamos de manera recursiva para lograr ejecutar en los subárboles de ser necesario.

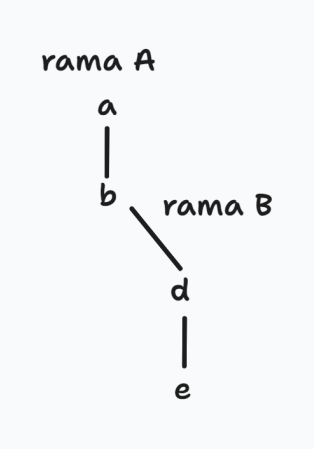
Por otro lado, modelamos el objeto commit, que si bien en un inicio no fue necesario, a la hora de realizar el rebase nos facilitó muchísimo el trabajo. Esto se debe a que decidimos que en vez de leer múltiples veces el archivo asociado al objeto commit en la carpeta gir/objects, directamente lo leemos una vez y guardamos toda la información que puede

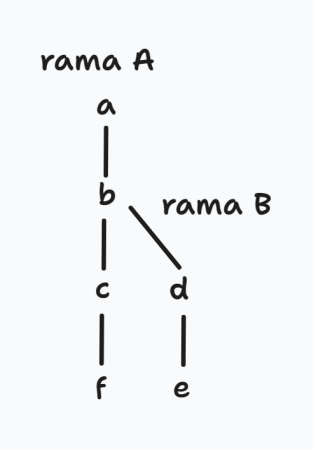
llegar a ser útil en un futuro. Es por esto que el objeto commit es algo complejo, ya que necesitamos una gran variedad de campos para guardar tanta información.

Detalles de la implementación

Merge

El merge es posiblemente de los comandos más complejos. Es por esto que vamos a dedicarle una sección completa para explicarlo. La primera tarea de este comando es buscar el último tree asociado a las dos branches que se quieren mergear, y teniendo eso, se busca el commit base entre ambas ramas. ¿Qué quiere decir esto? A continuación lo veremos con un diagrama.



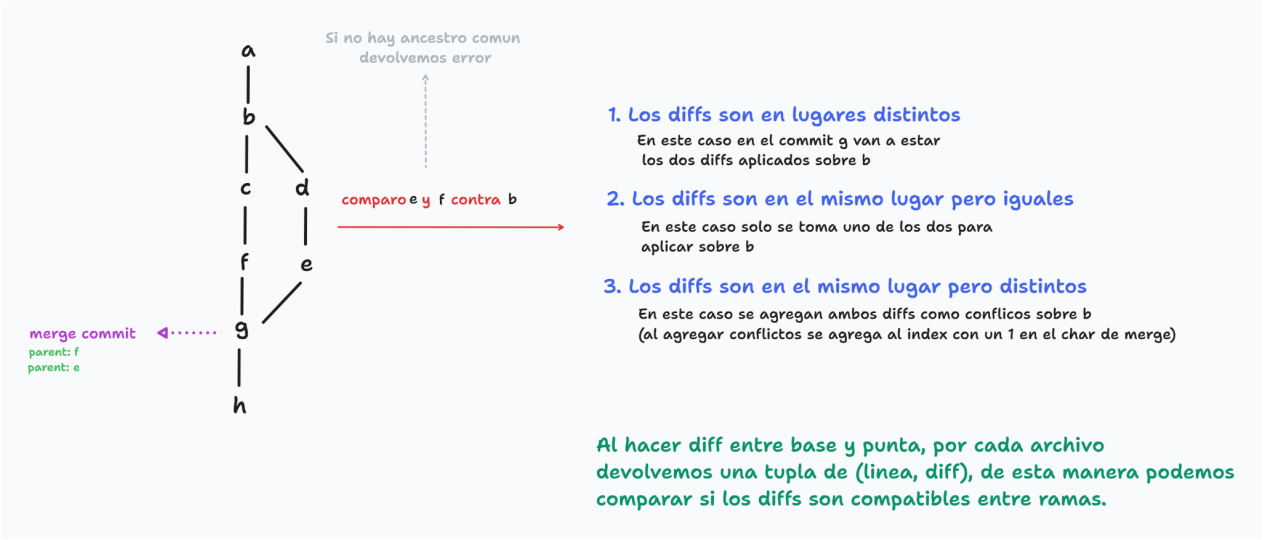


Con estas dos ramas de ejemplo, el commit base sería el primer commit que ambas ramas tienen en común, siendo en este caso B. Este commit es de suma importancia porque resulta ser la base a la cual cada rama calcula su diferencia con respecto al mismo. Cabe recalcar que en el caso de que el commit base sea igual al último commit de la branch actual se realizar un merge fast-forward que equivale a copiar el historial de commits desde la base al tip de la branch (en el ejemplo de la izquierda serían los commits d y e) y pegarlos en la branch actual.

Por otro lado, en el caso de un automerge tal como se puede observar en la imagen derecha, para calcular la diferencia entre un archivo y la base, primero calculamos las diferencias por línea de cada archivo con respecto a la versión del archivo en el commit base por medio del algoritmo de Longest Common Subsequence y en conjunto con otro algoritmo reconstruimos esas diferencias y las catalogamos según su tipo de modificación.

Luego, teniendo los diffs de ambas branches, las juntamos en un solo archivo marcando los conflictos en caso de que sucedan. Si suceden múltiples conflictos juntos, también unificamos los mismos para que en el display queden de una manera más amigable.

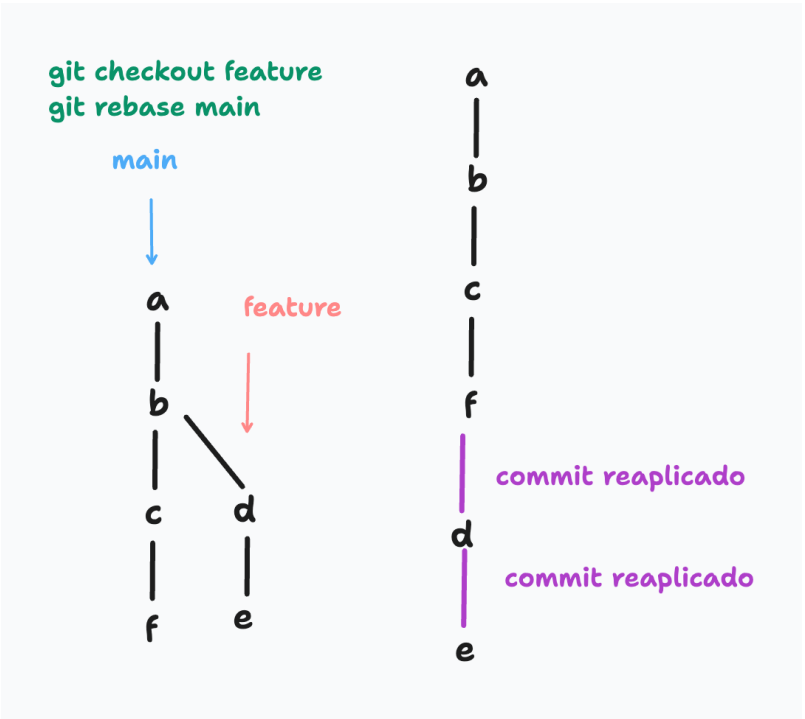
Repetimos este procedimiento con todos los archivos posibles del commit base y si no hay conflictos, el automerge finaliza y realiza un merge commit. Caso contrario, deja los archivos con conflictos en el index y pide al usuario que resuelva los conflictos.



Este diagrama refuerza la información mencionada anteriormente de una manera gráfica, demostrando a grandes rasgos cómo se ejecuta el automerge.

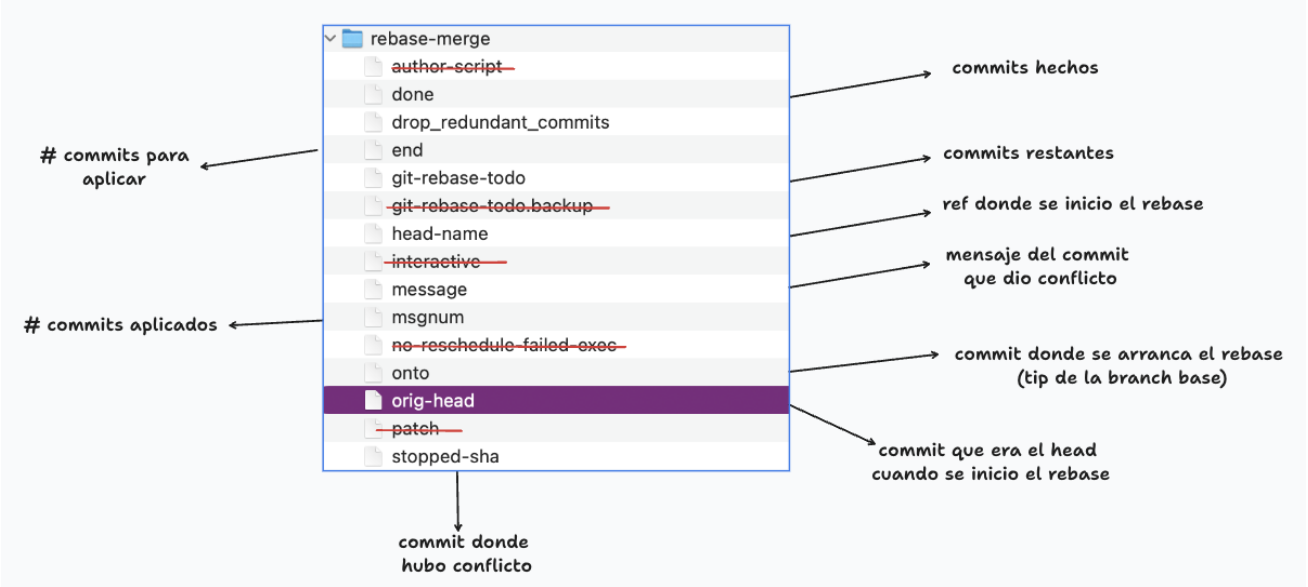
Rebase:

El rebase también fue complejo, sin embargo teniendo el merge ya cocinado, esta tarea fue más simple. El mismo consiste en buscar de la misma manera al padre en común como en el merge, pero ahora vamos a tratar un poco distinto a los commits. Aquí vamos a tomar todos los commits de la rama actual que estén por encima del padre común y luego vamos a ir aplicándolos uno por uno sobre la punta sobre la cual se está realizando el rebase. Tal cual como se muestra en el diagrama



Para poder aplicar cada commit, dentro del objeto commit creamos un método que es aplicar al directorio actual, el cual calcula el diff correspondiente al commit y luego intenta aplicarlo al mismo archivo en el directorio actual, si al aplicar un commit hay un conflicto, se termina esa instancia de rebase para que el usuario arregle el conflicto y continue con el rebase.

Para poder continuar con el rebase decidimos apegarnos al sistema que utiliza git, el cual puede ser visualizado en el diagrama, ignorando aquellos archivos que consideramos que no nos iban a ser útiles, las mismas están tachadas en rojo.

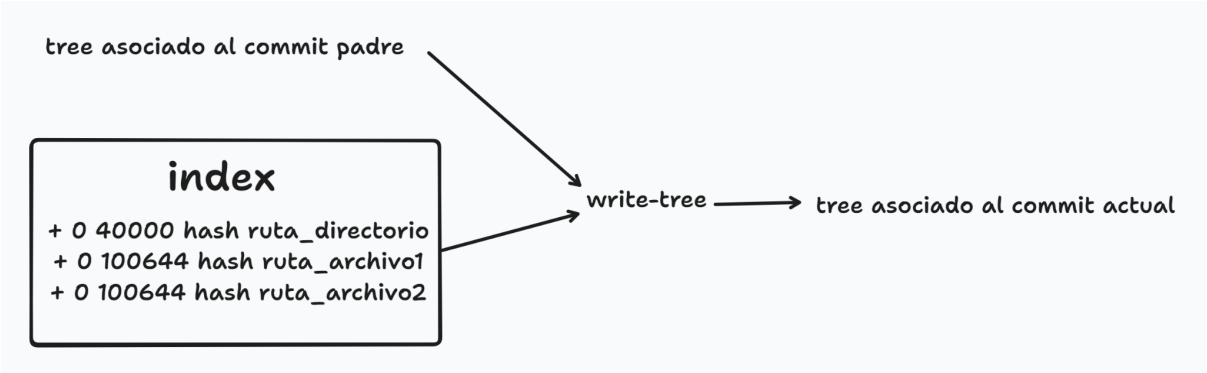


Write-tree:

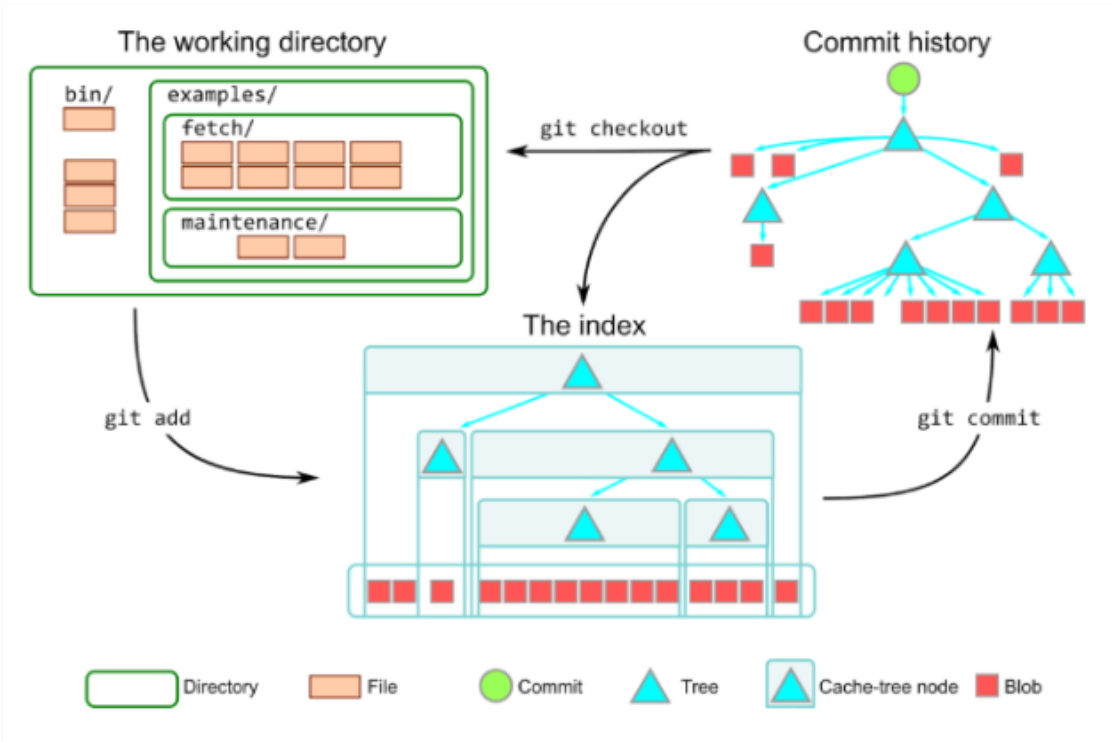
Otro comando de suma importancia para encapsular el creado de árboles previamente a commitear. Este se encarga de verificar las versiones del commit padre y las presentes actualmente y determina cuales deben estar presentes en el commit que se quiere realizar.

Para ello, primero en caso de ser el primer commit, toma todos los archivos que estén en la zona de staging. En otro caso, cuando hay un commit previo, hay que mantener las referencias previas pero, hay que verificar de sobreescribir los mismos en caso de que en la zona de staging justo exista una versión más actual.

Para lograr esto, utilizamos un hashmap con clave la ruta del archivo y valor del objeto. Primero agregamos todos los objetos del commit padre y luego los objetos presentes en el index. De esta manera, el hashmap solo se va a encargar de pisar la versión del commit anterior, ya que al insertarse por dirección, que en ambas versiones es la misma, como insertamos últimos los objetos del index, prevalecen las versiones del index.



En este diagrama se puede observar el proceso de creado del commit tree mencionado anteriormente.



A su vez, decidimos incluir este diagrama ya que en su momento nos fue de gran utilidad para entender cómo funcionan internamente algunos de los comandos de git. Muestra más cómo son las interacciones del cliente dentro de su directorio y repositorio local.

Logger

El logger nos fue de gran ayuda ya que este sirve para registrar información crucial sobre el funcionamiento del sistema en tiempo real. Este se encarga de llevar un registro de los eventos de cualquier índole (sea servidor o cliente). Estos eventos se registran por fecha y horario del evento en conjunto con un mensaje descriptivo de lo sucedido en ese timestamp. Este registro cumple diversas funciones, como facilitar el debugging y la corrección de errores en nuestro código.
Para implementar el logger sin tener un file-handler global, optamos por spawnear un thread por cada vez que se quería loguear, y mandarle un mensaje al thread interno del logger a

través de un channel, evitando así tener el error del file-handler global, además aportando flexibilidad y encapsulamiento.

Aprendizajes y conclusiones

El trabajo fue desafiante desde el principio, la mayoría del desafío vino de entender documentación escueta e imprecisa y adaptarla a las necesidades, pero también hubo un proceso de adaptación al ritmo de trabajo y las prácticas mismas, ya que la mayoría del grupo no contaba con experiencia en proyectos de esta envergadura, y mucho menos al estilo de colaboración propuesto. Más allá de eso, todos los participantes del grupo colaboraron de forma acorde con el proyecto, siguiendo las prácticas recomendadas por la cátedra, lo que facilitó en gran medida el desarrollo del mismo. En cuanto a GIT, se entendió mucho más de su comportamiento general, sacándole el velo “místico” que puede parecer tener en un principio. También se pudo comprender el uso de threads, y el protocolo ayudó a entender gran parte de las comunicaciones. A medio plazo, se comprendió la importancia del logger para debuggear de una manera eficiente y sin llenar todo de prints.

Pero en resumen podemos concluir que el mayor aprendizaje que nos llevamos del mismo es el aprendizaje de git. El tener que usarlo a nivel usuario para el desarrollo del trabajo como el tener que investigar e indagar sobre las funciones del mismo para implementarlo realmente ayudó a una gran entendimiento.