Proyecto Git Rústico

Grupo Rusteam Visionary

Integrantes

Valentina Lanzillotta

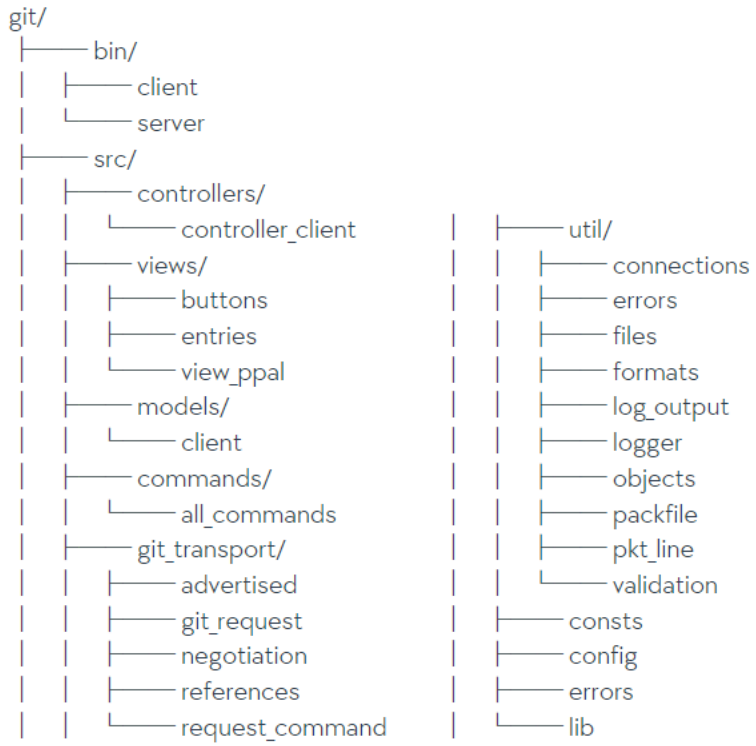
Brayan Ricaldi Rebata

Juan Sebastián Del Rio

Fecha de entrega: 4 diciembre de 2023

“Desarrollando un cliente y servidor Git desde cero en Rust, estamos construyendo las bases para una experiencia de control de versiones eficiente y segura”

Módulos del proyecto



Commands: En este módulo está la implementación de cada comando requerido para el funcionamiento del controlador de versiones.

git_transport: Este módulo contiene funciones que se usan para el protocolo git_transport. Como:

- Git request: Manejador de las solicitudes admitidas por el protocolo.
- Negotiation: Escritura y lectura de las referencias a negociar.
- References: Estructura que se usa para simular referencias, contienen su path y su tipo que puede ser tag, header, remote.

util: Este módulo contiene varias funciones que son útiles en todo el proyecto.
Como:

- La lectura/escritura de líneas en formato pkt.
- La lectura y envío de los packfile.
- Funciones que manejan archivos.
- Validaciones de los valores del Config.
- Creación del formato de los objetos (Commit,Tree, Blob,Tag)

MVC: Implementa la interacción del usuario a través de una interfaz gráfica (con la creación de botones, etiquetas, etc) para poder realizar las operaciones de los comandos Git

Comandos

- ## hash-object

El objetivo principal de “hash-object” es generar el hash de un objeto.
La implementación incluye manejo de argumentos, lectura de archivos y generación de hashes.
Funciones Principales:

handle_hash_object:

- Encargada de llamar a la función git_hash_object_blob con los parámetros adecuados, según la cantidad y tipo de argumentos proporcionados.
- Maneja diferentes casos en función de la cantidad y tipo de argumentos pasados.

git_hash_object_blob:

- Calcula el hash de un objeto de tipo blob utilizando el contenido de un archivo.
- Lee el contenido del archivo, prepara los datos del objeto y genera el hash.
- Utiliza funciones auxiliares como open_file, read_file, y hash_generate.

- ## cat-file

El propósito de este comando es mostrar el contenido o información sobre los objetos (blobs (archivos), commits, etc.) en un repositorio Git.
Funciones Principales:

handle_cat_file:
Función principal que maneja la ejecución del comando cat-file, validando los argumentos y llamando a la función git_cat_file.

git_cat_file:
Función que lee el contenido de un objeto en el repositorio Git (path del cliente dado) y realiza acciones según la flag especificada recibida en el handle (-t, -p, -s), type, content y size respectivamente.

git_cat_file_p:
Función que procesa y lee el contenido de un objeto según su tipo (blob, commit, tree, tag).

- init

El objetivo principal de estas funciones es inicializar (dado el path del cliente) un nuevo repositorio Git, creando los directorios y archivos necesarios. La implementación incluye la creación de directorios y archivos, y generación de un mensaje informativo sobre la inicialización del repositorio.

Funciones Principales:

handle_init:

- Encargada de llamar a la función git_init con los parámetros adecuados.
- Verifica que no se proporcionen argumentos adicionales (args.is_empty()).
- Retorna un mensaje informativo sobre la inicialización del repositorio.

git_init:

- Inicia un nuevo repositorio Git creando los directorios y archivos necesarios.
- Utiliza funciones auxiliares como create_directory y create_file pertenecientes al modulo Util.
- Estructura el repositorio con directorios como .git, .git/objects, .git/refs/heads, etc.
- Crea archivos importantes como .git/HEAD, .git/index, y .git/config.
- Retorna un mensaje informativo sobre la inicialización del repositorio.

- status

Estructuras de Datos: Se define una estructura StatusData para almacenar las listas de archivos modificados, no rastreados, en el área de preparación y eliminados.
Funciones Principales:

handle_status:

- Función principal que invoca la lógica de git_status.
- Verifica si hay argumentos adicionales y devuelve un error si los hay.

git_status:

- Utiliza varias funciones auxiliares para obtener información sobre el estado del repositorio.
- Calcula las diferencias (hashes) entre el directorio de trabajo y el área de preparación.
- Genera un vector detallado del estado del repositorio.

get_head_branch y get_index_content:

- Obtiene el nombre de la rama actual y el contenido del archivo de índice, respectivamente.

print_changes, branch_with_untracked_changes, branch_with_untracked_files, y branch_missing_commits:

- Formatean y muestran la salida del estado del repositorio.
- Proporcionan información sobre archivos modificados, no rastreados, en el área de preparación y eliminados.

is_files_to_delete y is_files_to_commit:

- Verifican si hay archivos para eliminar o confirmar, respectivamente.

compare_hash_lists y check_for_deleted_files:
- Comparan listas de hash para determinar los archivos modificados, no rastreados y eliminados.

check_for_commit, get_hashes_index, get_files_in_commit, y get_tree_content:
Se encargan de verificar si hay archivos en el área de preparación que deben confirmarse.
get_hashes_working_directory y calculate_directory_hashes:
Calculan los hashes de los archivos en el directorio de trabajo.
create_hash_working_dir:
Crea el hash de un archivo y lo agrega a la lista de hashes.

- add

Implementa la funcionalidad de agregar archivos al área de preparación (index) en un repositorio. El código maneja la adición de archivos específicos y la adición de todos los archivos en el directorio ".".
Funciones Principales:

handle_add:
Función principal que maneja la ejecución del comando add. Puede agregar un archivo específico o todos los archivos en el directorio utilizando un punto.
git_add_all:
Agrega todos los archivos del directorio al área de preparación.
add_file:
Agrega un archivo específico al área de preparación.
git_add:
Maneja la lógica principal de agregar un archivo al área de preparación, incluida la creación de objetos utilizando el módulo util, y actualización del index.

- rm

El código implementa la funcionalidad de eliminar archivos del working directory y del index.
Funciones Principales:

handle_rm:
Función que toma los argumentos y el cliente, verifica la cantidad de argumentos y luego llama a git_rm.
git_rm:
Función que realiza la eliminación de un archivo del working directory y del index. Llama a compare_hash para comparar el hash del archivo en el working directory con el del index.
compare_hash:
Compara el hash del archivo que se quiere remover con el hash de ese archivo en el index. Luego, llama a remove_from_index para eliminarlo del index y del working directory si corresponde.

remove_from_index_with_filename:
Elimina el archivo del index utilizando su nombre.

remove_from_index:
Elimina el archivo del index y del working directory si corresponde.
update_index:
Actualiza el archivo de index con las líneas proporcionadas.

- commit

El módulo commit se encarga de la creación de commits, la gestión de archivos relacionados con los commits, y la construcción de objetos de commit en un repositorio Git local.
Funciones Principales:

handle_commit:
Función principal que maneja la ejecución del comando commit, validando los argumentos y llamando a git_commit.
git_commit:
Función que realiza el commit, construye el objeto commit, actualiza el índice y el archivo (hash) de la branch actual.

- checkout

Se encarga de cambiar entre ramas y crear nuevas ramas (-b). Además, realiza la carga de archivos asociados con la rama seleccionada y maneja escenarios de error, como la existencia de cambios sin confirmar.
Funciones Principales:

handle_checkout:
Función principal que maneja la ejecución del comando checkout, validando los argumentos y llamando a git_checkout_switch.
git_checkout_switch:
Función que realiza la operación principal de cambiar de rama o crear una nueva rama (-b). También gestiona la carga de archivos asociados con la rama seleccionada y maneja cambios sin confirmar.

- log

El propósito principal de estas funciones es mostrar el historial de commits en un formato legible. Se utilizan operaciones de lectura de archivos, formato de fechas y manipulación de cadenas para presentar la información de log de manera estructurada.

handle_log:
  - Encargada de llamar a la función git_log con los parámetros adecuados.
  - Verifica que no se proporcionen más de un argumento.
  - Retorna el resultado de git_log.

git_log:
  - Construye la ruta al archivo de log utilizando el nombre de la rama actual.

- Abre el archivo de log y procesa sus líneas.
- Llama a la función get_parts_commit para formatear las partes relevantes del commit.
- Retorna el resultado formateado del log.

get_parts_commit:
- Extrae partes específicas de las líneas del archivo de log para formar un resultado formateado.
- Convierte la marca de tiempo del commit a un formato de fecha legible.
- Retorna un resultado formateado del log.

Auxiliares:
- insert_line_between_lines: Inserta una línea en una cadena entre líneas específicas.
- save_parent_log: Recorre los padres de los commits y registra en el log de la rama.
- save_log: Guarda los logs de los commits recibidos del servidor.

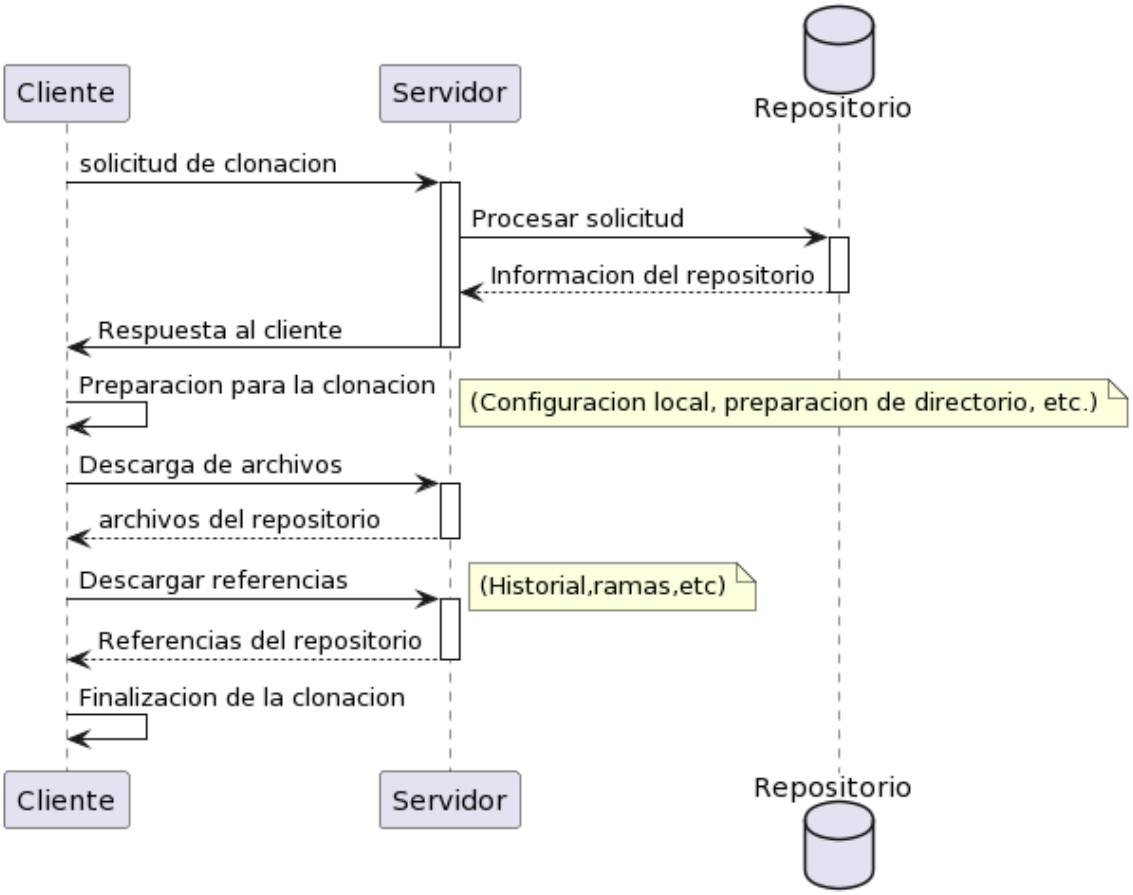
• clone

Se encarga de clonar un repositorio Git desde un servidor remoto utilizando el protocolo Git (se detalla más adelante) . Además, maneja la construcción y la organización de los objetos del repositorio clonado en el directorio local. El código utiliza sockets TCP para la comunicación con el servidor Git remoto.

- **handle_clone:**
  Función principal que maneja la ejecución del comando clone, validando los argumentos y llamando a git_clone.
- **git_clone:**
  Función que realiza la clonación del repositorio, maneja la negociación de paquetes, recibe el contenido del packfile y organiza los objetos en el directorio local.

Funciones Auxiliares:
- **create_repository:**
  Crea un repositorio local a partir de los objetos recibidos del servidor.
- **recovery_blob:**
  Construye el blob.
- **recovery_tree:**
  Recorre los objetos sub-tree, los construye y los agrega al índice.
- **save_references:**
  Guarda las referencias recibidas del servidor en el repositorio local.
- **handle_commit**:
  Construye el objeto Commit recibido del servidor.
- **handle_tree**:
  Recorre los objetos tree, los construye y los agrega al índice.

Diagrama de secuencia del comando



- fetch

Se utiliza para descargar objetos y referencias desde otro repositorio.

FetchStatus:

- Una enumeración que representa diferentes estados de la operación de fetch. Proporciona mensajes legibles para diferentes escenarios.

handle_fetch:

- La función principal para manejar el comando fetch. Recibe los argumentos del comando y un objeto Client que representa la configuración del cliente Git.
- Distingue entre el fetch de todos los remotos (git fetch) y el fetch de una rama específica (git fetch <rama>).

git_fetch_all:

- Realiza el fetch de actualizaciones desde todos los remotos especificados en la configuración Git.
- Itera sobre cada remoto, realiza una serie de operaciones de fetch de Git y devuelve un resumen del estado del fetch.

git_fetch_branch:

- Maneja la operación de fetch para una rama específica.

- Valida la existencia de la rama en el repositorio local.
- Realiza el descubrimiento de referencias, la negociación del packfile y recupera datos del packfile.
- Actualiza referencias locales y guarda objetos y referencias recibidos.

- merge

Funciones Principales:

handle_merge:
- Verifica la cantidad correcta de argumentos y luego llama a git_merge.
- Recibe un vector de argumentos y un objeto Client.

git_merge:
- Llama a try_for_merge y maneja el resultado.
- Recibe el directorio del repositorio, la rama actual, el nombre de la rama a fusionar y un objeto Client.

try_for_merge:
- Intenta fusionar la rama actual con otra rama.
- Maneja diferentes estrategias de fusión: fast-forward o recursive.
- Actualiza referencias y logs según sea necesario.
- Recibe el directorio del repositorio, la rama actual, el nombre de la rama a fusionar, un objeto Client y el tipo de fusión.

Otras Funciones de Utilidad:
- Varias funciones de utilidad para actualizar referencias, manejar conflictos, actualizar logs, etc.

- remote

Funciones Principales:

handle_remote:
Función que toma los argumentos y el cliente, verifica la cantidad de argumentos y luego llama a git_remote.

git_remote:
Ejecuta la acción remota en el repositorio local. Puede mostrar los remotos existentes, agregar un nuevo remoto o eliminar un remoto existente.

get_remotes:
Obtiene los nombres de los remotos existentes desde el contenido del archivo de configuración.

add_remote:
Agrega un nuevo repositorio remoto al archivo de configuración. Verifica si el remoto ya existe antes de agregarlo.

check_if_remote_exists:
Verifica si un remoto ya existe en el archivo de configuración.

remove_remote:
Elimina un repositorio remoto del archivo de configuración.

- pull

Funciones Principales:

handle_pull:
- Maneja la operación de "pull".
- Valida la cantidad correcta de argumentos.
- Inicia una operación de "pull" desde el servidor de Git.
- Actualiza la configuración del repositorio local si se especifica un repositorio remoto y una rama remota.
- Llama a git_pull para realizar el "pull" real.
- Devuelve un resultado que indica el éxito o un error encontrado durante la operación de "pull".

git_pull:
- Realiza un "fetch" del repositorio remoto usando git_fetch_branch.
- Verifica si hay actualizaciones que necesitan ser fusionadas.
- Obtiene la referencia remota de la rama actual.
- Realiza un "merge" con el repositorio remoto utilizando git_merge.
- Maneja conflictos si ocurren.
- Actualiza el archivo FetchHead y otros estados según sea necesario.
- Devuelve un resultado que indica el éxito o un error encontrado durante la operación de "pull".

- push

Estructura PushBranch: Almacena información necesaria para la operación "push", como la ruta local, el nombre remoto, la URL remota, la configuración Git, la rama y el estado. Implementa métodos para inicializar el estado, agregar mensajes al estado, obtener el estado, obtener el hash de la rama y la ruta local.

handle_push:
- Maneja la operación "push".
- Valida la cantidad correcta de argumentos.
- Inicia una operación "push" en el servidor Git.
- Actualiza la configuración del repositorio local si se especifica un repositorio remoto y una rama remota.
- Llama a git_push_branch para realizar la operación "push" real.
- Devuelve un resultado que indica el éxito o un error encontrado durante la operación "push".

git_push_branch:
- Prepara la solicitud "git-upload-pack" para el servidor.

- Realiza la referencia remota al servidor y obtiene el hash previo de la rama remota.
- Determina si es necesario realizar la operación de actualización en el servidor remoto.
- Actualiza la referencia en el servidor Git con los hashes de commits proporcionados.
- Envía los objetos que no tiene el servidor remoto.
- Devuelve un resultado que indica el éxito o un error encontrado durante la operación "push".

• branch

El código permite listar, crear y eliminar ramas en un repositorio local. También proporciona funciones para obtener información sobre la rama actual, obtener el hash de una rama remota, y copiar el historial de una rama a otra.
Funciones Principales:

handle_branch:
Función principal que maneja la ejecución del comando branch, permitiendo listar, crear y eliminar ramas según los argumentos proporcionados.
git_branch_list:
Lista las ramas existentes y marca la rama actual con un asterisco (*).
git_branch_create:
Crea una nueva rama, copiando el contenido y el historial de la rama actual.
git_branch_delete:
Elimina una rama existente (no puede eliminar la rama actual).
get_current_branch:
Obtiene el nombre de la rama actual.
get_branch_remote:
Obtiene las ramas remotas de un repositorio.
get_branch:
Obtiene las ramas locales de un repositorio.
copy_log:
Copia el historial de commits de una rama a otra.

• check-ignore

Este comando verifica si los archivos o directorios proporcionados como argumento están incluidos en el archivo .gitignore.
El código tiene soporte para leer el contenido de **.gitignore** y evitar agregar archivos que están excluidos por patrones en el archivo.
Funciones Principales:

handle_check_ignore:

Función principal que maneja la ejecución del comando check-ignore, validando los argumentos y llamando a la función git_check_ignore.

git_check_ignore:

Función que verifica si los archivos o directorios pasados como parámetro están incluidos en el archivo .gitignore.

get_gitignore_content:

Obtiene el contenido del archivo .gitignore.

check_gitignore:

Verifica si un path está incluido en .gitignore.

- ls-files

handle_ls_files, verifica los argumentos y llama a git_ls_files para realizar la funcionalidad específica de listar archivos en diferentes situaciones.

Funciones Principales:

handle_ls_files:

- Encargada de llamar a la función git_ls_files con los parámetros adecuados.
- Verifica la validez de los argumentos, incluyendo el número correcto de argumentos y si las flags son reconocidas.
- Retorna el resultado de git_ls_files.

git_ls_files:

- Lista archivos en diferentes contextos según la flagespecificada.
- Llama a funciones auxiliares (get_deleted_files y get_modified_or_untracked_files) según la flag.
- Retorna el resultado formateado.

get_deleted_files:

- Compara los hashes de los archivos en el índice y el directorio de trabajo para encontrar archivos eliminados.
- Retorna un resultado formateado con los archivos eliminados.

get_modified_or_untracked_files:

- Compara los hashes de los archivos en el índice y el directorio de trabajo para encontrar archivos modificados o no rastreados.
- Retorna un resultado formateado con los archivos modificados o no rastreados.

- ls-tree

handle_ls_tree, verifica los argumentos y llama a git_ls_tree para realizar la funcionalidad específica de listar el contenido de un árbol (tree-ish).

Funciones Principales:

handle_ls_tree:
- Encargada de llamar a la función git_ls_tree con los parámetros adecuados.
- Verifica la validez de los argumentos, incluyendo el número correcto de argumentos.
- Retorna el resultado de git_ls_tree.

git_ls_tree:
- Lista el contenido de un árbol (tree-ish) especificado.
- Utiliza funciones auxiliares para obtener información asociada al árbol.
- Retorna el resultado formateado.

associated_commit:
- Obtiene el commit asociado a un árbol (tree-ish) especificado.
- Utiliza otras funciones auxiliares (associated_tree y get_head_tree).
- Retorna el hash del árbol asociado al commit.

associated_tree:
- Obtiene el árbol asociado a un commit hash.
- Utiliza la función git_cat_file.
- Retorna el hash del árbol asociado.

get_head_tree:
- Obtiene el árbol asociado a HEAD.
- Utiliza la función associated_tree.
- Retorna el hash del árbol asociado.

- show-ref

Implementa la funcionalidad para mostrar las referencias (refs) de un repositorio local con sus commits.

Funciones Principales:

handle_show_ref:
Función que toma los argumentos y el cliente, verifica la cantidad de argumentos y luego llama a git_show_ref.

git_show_ref:
Muestra las referencias de un repositorio local con sus commits. Llama a visit_refs_dirs para recorrer los directorios de .git/refs y agregar los contenidos al resultado formateado.

visit_refs_dirs:
Recorre los directorios de .git/refs y agrega los contenidos al resultado formateado. Utiliza la recursión para manejar subdirectorios.

- rebase

Funciones Principales:

handle_rebase:

Función que toma los argumentos y el cliente, verifica la cantidad de argumentos y luego llama a git_rebase.
git_rebase:
Realiza el rebase de una rama sobre otra. Obtiene los hashes de los commits de ambas ramas, realiza un intento de fusión (merge), y si tiene éxito, crea nuevos commits para actualizar el log.
update_first_commit:
Actualiza el primer commit de la rama actual para que el padre sea el último commit de la rama sobre la cual se hizo el rebase.
update_parent:
Actualiza el padre del primer commit de la rama actual con el último commit de la rama sobre la cual se hizo el rebase.
rewrite_commit:
Reescribe el commit con el padre actualizado.
create_new_commits:
Crea nuevos commits por cada commit que está en la rama actual y no en la rama sobre la cual se hizo el rebase.
update_log_current_branch:
Actualiza el log de la rama actual con el log de la rama sobre la cual se hizo el rebase.
get_commit_msg: Obtiene el mensaje de un commit.

- tag

Manejo de etiquetas (create-delete-list) . Resumen de las principales funciones y su propósito:

handle_tag:
Función principal que maneja la lógica principal para las operaciones de etiquetas. Puede realizar acciones como mostrar todas las etiquetas, crear una nueva etiqueta anotada y eliminar una etiqueta existente.
get_tags:
Obtiene la lista de nombres de todas las etiquetas en el repositorio local.
git_tag:
Muestra todas las etiquetas existentes en el repositorio.
builder_tag_msg_edit:
Crea un archivo para almacenar el mensaje asociado con una etiqueta.
git_tag_create:
Crea una nueva etiqueta anotada. Genera un objeto de etiqueta que incluye información sobre el compromiso asociado.

git_tag_delete:
Elimina una etiqueta existente.

Manejo de errores

El código maneja errores personalizados mediante el uso del tipo CommandsError, UtilError, GitError, dependiendo el módulo en el que se encuentre, proporcionando información detallada sobre errores específicos.

Objetos

Al igual que el auténtico Git, nuestro programa utiliza la misma estructura/formato de los objetos (Blobs, Commits, Tree y Tags)

Blob

- Encabezado: Contiene la palabra clave "blob", un espacio y el tamaño del contenido en bytes, seguido de un carácter nulo.
- Contenido: El contenido real del archivo.

Tree

- Encabezado: Contiene la palabra clave "tree", un espacio y el tamaño del contenido en bytes, seguido de un carácter nulo.
- Contenido: Una serie de entradas, cada una de las cuales contiene permisos, nombre del archivo o subdirectorio, un carácter nulo, y el hash SHA-1 del blob o tree correspondiente.

Commit

- Encabezado: Contiene la palabra clave "commit", un espacio y el tamaño del contenido en bytes, seguido de un carácter nulo.
- Contenido: Campos para el árbol (hash del árbol), padres (hash de los commits padres), autor, comisor, fecha, mensaje y otros metadatos.

  commit {tamaño}\0
  tree {hash}
  parent {hash}
  author {nombre} <{correo}> {fecha}
  committer {nombre} <{correo}> {fecha}

  {mensaje del commit}

Tag

- Encabezado: Contiene la palabra clave "tag", un espacio y el tamaño del contenido en bytes, seguido de un carácter nulo.

- Contenido: Campos para el tipo del objeto referenciado ("commit"), el hash del objeto referenciado, el autor, la fecha, y el mensaje de la etiqueta.

  tag {tamaño}\0
  object {tipo} {hash}
  type {tipo}
  tag {nombre del tag}
  tagger {nombre} <{correo}> {fecha}
  {mensaje de la etiqueta}

Patrón Model-View-Controller

Controlador (controller_client)

- Define un controlador que interactúa con el Cliente y la interfaz.
- Implementa funciones para enviar comandos Git (con sus flags), obtener información del cliente (nombre, correo electrónico, directorio actual, rama actual, etc.).
- Utiliza un conjunto de comandos Git implementados en diferentes módulos.

Modelo (Client)

- Representa la información del cliente, incluyendo nombre, correo electrónico, dirección IP, puerto y el path del repositorio.
- Proporciona métodos para acceder y actualizar la información del cliente.

Dicha información se obtiene del archivo de configuración que será detallado más adelante en el presente informe.

Vista (Glade)

- Utiliza GTK (glade) para crear la interfaz de usuario.
- Define funciones para establecer etiquetas, manejar eventos de clic en botones y conectar la interfaz con el controlador.
- Implementa una ventana principal y ventanas de diálogo para diferentes comandos Git.
- Se utilizan ventanas de diálogo separadas para comandos específicos de Git como clone, cat-file, hash-object, fetch, push y pull.
- La vista contiene etiquetas para mostrar información como nombre de usuario, correo electrónico, rama actual y directorio de trabajo.
- Los botones de la interfaz están conectados a funciones que envían comandos Git al controlador.

Se utiliza un patrón MVC (Modelo-Vista-Controlador) para organizar la lógica del programa.

Diagrama de clases del patrón MVC

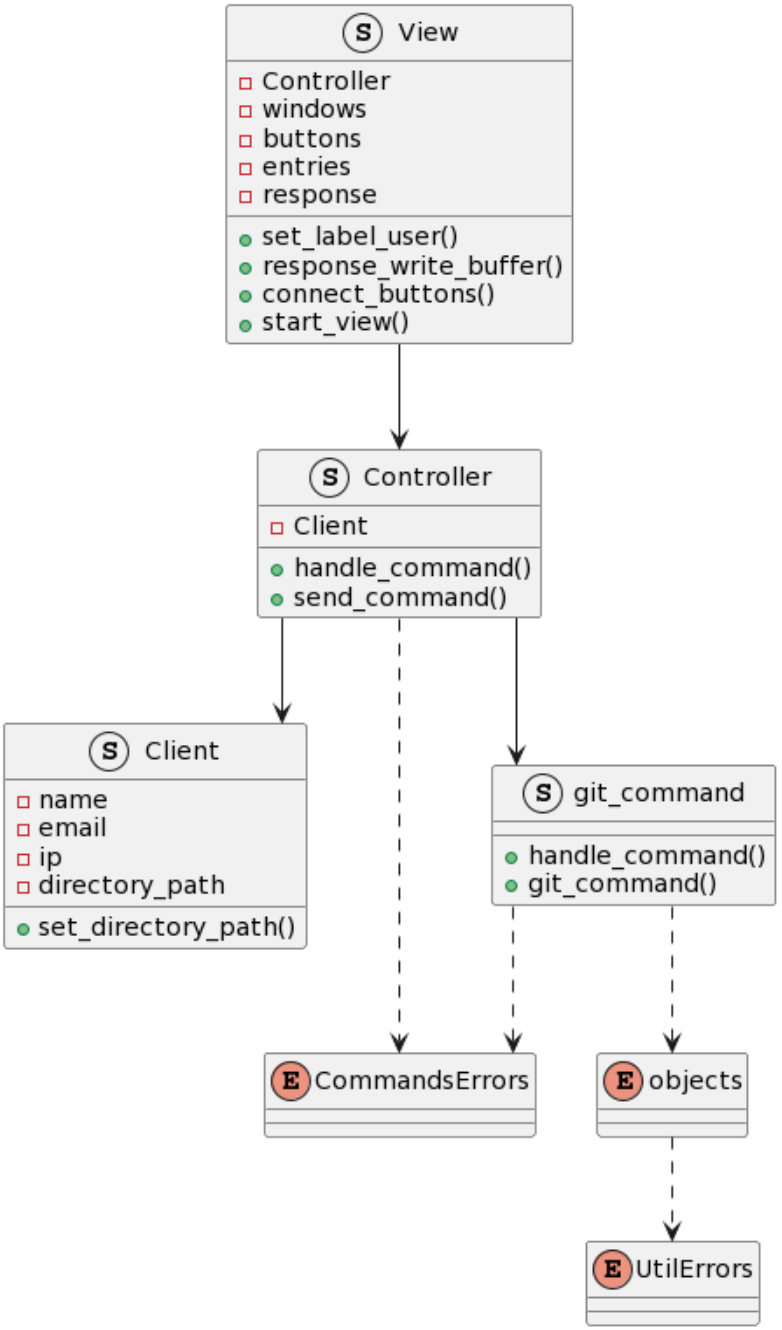
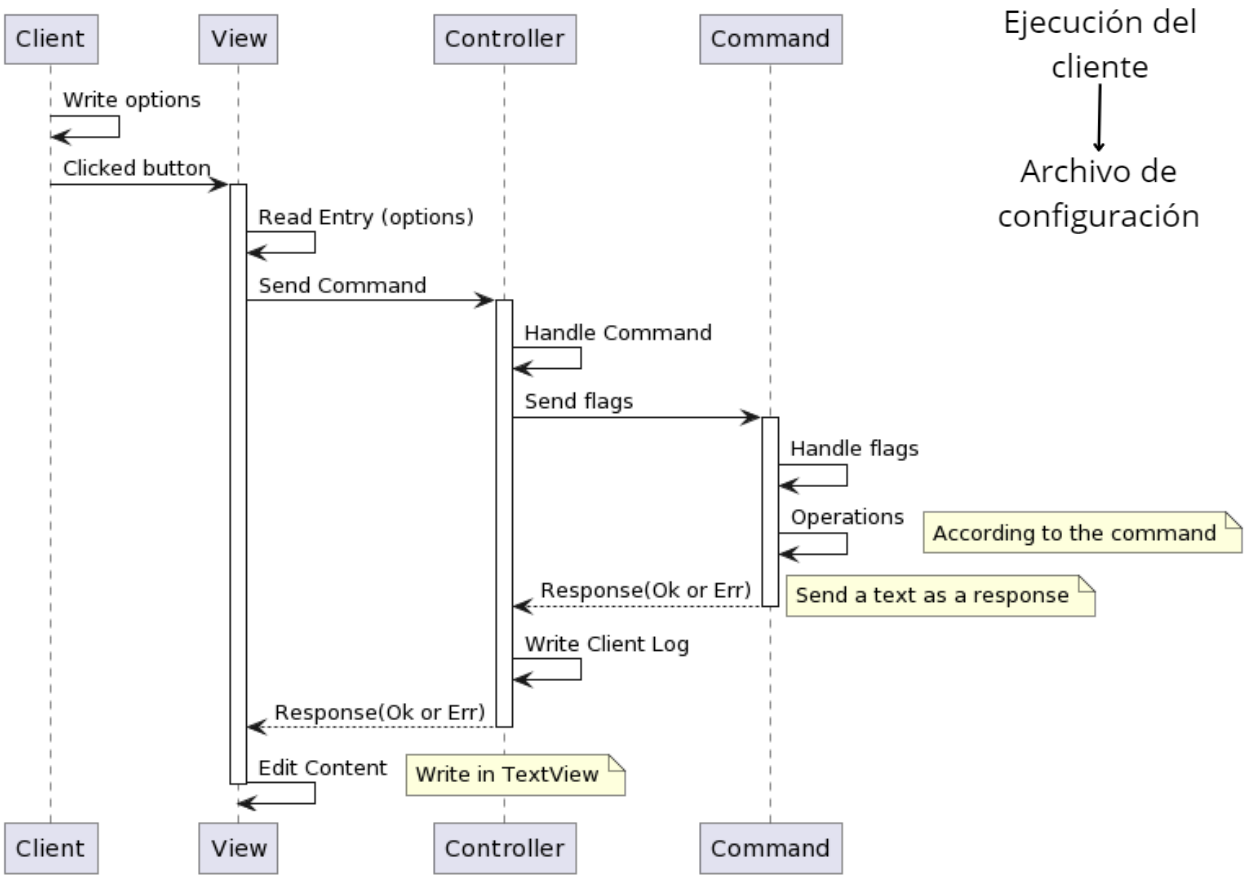


Diagrama de secuencia del patrón MVC



Archivo de configuración

Estructura del Archivo de Configuración

El archivo de configuración sigue una estructura de clave-valor. Cada línea contiene un par de la forma clave=valor.

Claves y Valores

- Name: Nombre descriptivo del servidor/cliente.
- Email: Dirección de correo electrónico asociada al servidor/cliente.
- Path log: Ruta del archivo de historial o registro del servidor/cliente.
- Ip: Dirección del servidor.
- Port: Puerto en el cual el servidor escuchará las conexiones.
- Src: En caso del servidor, el directorio raíz donde se guardarán los archivos. En caso del cliente, donde está nuestro repo git o la carpeta donde se guardará nuestro repo si se clona.

Validaciones

Cada clave tiene una función de validación del valor. En el caso de la ip se asegura que sea una IP v4 o v6. Si fuera v4 se valida cada octeto, si fuera v6 se valida de igual manera cada campo.

Uso en el programa

Los ejemplos de uso son los siguientes:

- cargo run --bin client -- path_config_cliente
- cargo run --bin server -- path_config_server

Servidor

El Servidor permitirá al cliente descargar el contenido del repositorio respetando la implementación de upload-pack y recibir actualizaciones de contenido por parte de los clientes mediante receive-pack siguiendo el protocolo Git Transport. Estas operaciones serán utilizadas por el cliente para implementar los comandos que interactúan con el Servidor, como por ejemplo fetch, pull, push, etc.

Upload-pack

El "upload-pack" es una parte esencial del protocolo de transferencia de Git que facilita la eficiente transferencia de datos desde el servidor Git al cliente Git durante operaciones como "git fetch". Este proceso ocurre en el lado del servidor.

Aquí hay una explicación detallada de cómo funciona el "upload-pack" en el contexto del protocolo de transferencia de Git:

Client request

Cuando un cliente Git inicia una operación de upload-pack(clone o fetch), envía una solicitud al servidor Git remoto para obtener actualizaciones. Esta solicitud incluye el la información del server como la ip y el puerto y la información del repositorio del cual se quiere obtener la solicitud.

Reference Discovery

El servidor responde proporcionando información sobre las referencias disponibles en el repositorio remoto. Esto puede incluir ramas, etiquetas y otras referencias.

Packfile Negotiation

El cliente y el servidor comparan las referencias locales y remotas para determinar qué objetos son necesarios para que el cliente se actualice.
Se acuerda una lista de objetos que se enviarán al cliente para que su repositorio local esté actualizado. Esta negociación garantiza que solo se envíen los objetos necesarios y no se desperdicie ancho de banda.
Si el cliente no confirma que tiene referencias, entonces se hará un clone. Si el cliente cuenta con referencias disponibles, no importa si están atrasadas, será un fetch.

Packfile Data

Con la lista de objetos acordada, el servidor crea un "packfile" que contiene esos objetos. El "packfile" se envía al cliente como respuesta a la solicitud upload-pack.
El cliente recibe el "packfile", lo descomprime y extrae los objetos necesarios, actualizando así su repositorio local.

Receive-pack

El receive-pack es una operación crucial en el protocolo Git y ocurre cuando un cliente realiza un git push hacia el servidor. A continuación, se presenta un resumen detallado desde el punto de vista del servidor:

Client request y Reference Discovery

Se inicia con una petición del cliente y reference discovery como se explicó en upload-pack.

Reference Update Request

El servidor recibirá listas de referencias con el hash de la referencia actual y el hash de la nueva referencia.
Al finalizar el listado de las referencias a actualizar se recibirá un mensaje FLUSH dado inicio a la lectura del “packdata”.

Packfile Transfer

Se leerá el “packdata” de solo los objetos coordinados en la etapa Reference Update Request, así solo se recibirá lo mínimo.

Report Status

Al finalizar la lectura de los datos y actualizar las referencias de nuestro repositorio remoto, se enviará un mensaje estatus por cada referencia actualizada.
Comunicando si se pudo actualizar la referencia y si no, explicando el motivo.

Crate externos

Se utilizaron los siguientes crates externos:

- **chrono:** para la obtención del timestamp actual.
- **sha1**: para la función de hash SHA1.
- **flate2**: para comprimir y descomprimir contenidos.
- **gtk-rs:** para la implementación de la interfaz gráfica.