```
# 2024 Â© Idan Hazay
# Import required libraries

import traceback, time, os
from modules import validity  # Import validation module for input checking
from modules.config import *  # Configuration settings
from modules.limits import Limits  # Subscription limits and restrictions
from modules.errors import Errors  # Error handling
from filelock import FileLock  # File locking to prevent concurrent access

class Protocol:
    """
    Handles the processing of client requests and generates appropriate responses.
    This class interprets incoming messages, validates data, and executes necessary actions.
    """

    def __init__(self, network, clients, cr, files_uploading):
        self.network = network  # Handles network communication
        self.clients = clients  # Stores active client sessions
        self.v = validity.Validation()  # Instance of validation class for input checks
        self.cr = cr  # Handles client database interactions
        self.files_uploading = files_uploading  # Tracks files currently being uploaded
        self.files_in_use = []  # Stores files that are currently being accessed

    def protocol_build_reply(self, request, tid, sock):
        """
        Parses client requests, validates input, and determines the appropriate response.
        Each request has an action code that specifies the intended operation.
        """
        if request is None:
            return None

        fields = request.split(b"|")  # Split the request into parts using "|"
        code = fields[0].decode()  # Extract the action code

        if code not in ["FILD", "FILE"]:  # If not a file-related request, decode to string
            fields = request.decode().split("|")

        if code == 'EXIT':
            reply = 'EXTR'  # Send exit confirmation to client
            self.clients[tid].id = tid  # Reset client ID
            self.clients[tid].user = "dead"  # Mark client as disconnected

        elif code == "LOGN": # Client login request: validate credentials and grant access
            cred, password = fields[1], fields[2]

            if self.v.is_empty(fields[1:]):  # Ensure fields are not empty
                return Errors.EMPTY_FIELD.value
            elif self.v.check_illegal_chars(fields[1:]):  # Check for illegal characters
                return Errors.INVALID_CHARS.value

            if self.cr.login_validation(cred, password):  # Validate credentials
                if not self.cr.verified(cred):  # Ensure account is verified
                    reply = Errors.NOT_VERIFIED.value
                else:
                    user_dict = self.cr.get_user_data(cred)  # Retrieve user data from database
                    self.clients[tid].id = user_dict["id"]
                    self.clients[tid].user = user_dict["username"]
                    self.clients[tid].email = user_dict["email"]
                    self.clients[tid].subscription_level = user_dict["subscription_level"]
                    self.clients[tid].admin_level = user_dict["admin_level"]
                    reply = f"LOGS|{user_dict['email']}|{user_dict['username']}|{int(user_dict['subscription_level'])}|
{self.clients[tid].admin_level}"
            else:
                reply = Errors.LOGIN_DETAILS.value  # Send error if credentials are incorrect

        elif code == "SIGU": # Client signup request: register new users
            email, username, password, confirm_password = fields[1:5]

            if self.v.is_empty(fields[1:]):
                return Errors.EMPTY_FIELD.value
            elif self.v.check_illegal_chars(fields[1:]):
                return Errors.INVALID_CHARS.value
            elif not self.v.is_valid_email(email):
                return Errors.INVALID_EMAIL.value
            elif not self.v.is_valid_username(username) or username == "guest":
                return Errors.INVALID_USERNAME.value
            elif not self.v.is_valid_password(password):
                return Errors.PASSWORD_REQ.value
            elif password != confirm_password:
                return Errors.PASSWORDS_MATCH.value

            if self.cr.user_exists(username):  # Check if username already exists
                reply = Errors.USER_REGISTERED.value
            elif self.cr.email_registered(email):  # Check if email is already used
                reply = Errors.EMAIL_REGISTERED.value
            else:
```

```
        self.cr.signup_user([email, username, password])  # Register new user
        self.cr.send_verification(email)  # Send verification email
        reply = f"SIGS|{email}|{username}|{password}"

elif code == "FOPS": # Request password reset code
    email = fields[1]

    if self.v.is_empty(fields[1:]):
        return Errors.EMPTY_FIELD.value
    elif self.v.check_illegal_chars(fields[1:]):
        return Errors.INVALID_CHARS.value
    elif not self.v.is_valid_email(email):
        return Errors.INVALID_EMAIL.value

    if self.cr.email_registered(email):
        if not self.cr.verified(email):
            reply = Errors.NOT_VERIFIED.value
        else:
            self.cr.send_reset_mail(email)  # Send password reset email
            reply = f"FOPR|{email}"
    else:
        reply = Errors.EMAIL_NOT_REGISTERED.value

elif code == "PASR": # Reset password after receiving verification code
    email, code, new_password, confirm_new_password = fields[1:5]

    if self.v.is_empty(fields[1:]):
        return Errors.EMPTY_FIELD.value
    elif self.v.check_illegal_chars(fields[1:]):
        return Errors.INVALID_CHARS.value
    elif not self.v.is_valid_password(new_password):
        return Errors.PASSWORD_REQ.value
    elif new_password != confirm_new_password:
        return Errors.PASSWORDS_MATCH.value

    res = self.cr.check_code(email, code)  # Validate reset code
    if res == "ok":
        self.cr.change_password(email, new_password)  # Update password in database
        self.clients[tid].user = "guest"
        reply = f"PASS|{email}|{new_password}"
    elif res == "code":
        reply = Errors.NOT_MATCHING_CODE.value
    else:
        reply = Errors.CODE_EXPIRED.value

elif code == "LOGU": # Client logout request
    self.clients[tid].id = tid
    self.clients[tid].user = "guest"
    self.clients[tid].email = "guest"
    self.clients[tid].subscription_level = 0
    self.clients[tid].admin_level = 0
    reply = "LUGR"  # Logout confirmation message

elif code == "SVER": # Resend verification email for unverified accounts
    email = fields[1]

    if self.v.is_empty(fields[1:]):
        return Errors.EMPTY_FIELD.value
    elif self.v.check_illegal_chars(fields[1:]):
        return Errors.INVALID_CHARS.value
    elif not self.v.is_valid_email(email):
        return Errors.INVALID_EMAIL.value

    if self.cr.email_registered(email):
        if self.cr.verified(email):
            reply = Errors.ALREADY_VERIFIED.value
        else:
            self.cr.send_verification(email)
            reply = f"VERS|{email}"
    else:
        reply = Errors.EMAIL_NOT_REGISTERED.value

elif (code == "VERC"):   # Client requests account verification
    email = fields[1]
    code = fields[2]

    if (self.v.is_empty(fields[1:])):
        return Errors.EMPTY_FIELD.value
    elif (self.v.check_illegal_chars(fields[1:])):
        return Errors.INVALID_CHARS.value
    elif (not self.v.is_valid_email(email)):
        return Errors.INVALID_EMAIL.value

    if (self.cr.email_registered(email)):
        res = self.cr.check_code(email, code)
        if (res == "ok"):
            self.cr.verify_user(email)
```

```
                self.cr.send_welcome_mail(email)
                reply = f"VERR|{email}"
            elif (res == "code"):
                reply = Errors.NOT_MATCHING_CODE.value
            else:
                reply = Errors.CODE_EXPIRED.value
        else:
            reply = Errors.EMAIL_NOT_REGISTERED.value

    elif code == "DELU": # Client requests account deletion
        email = fields[1]
        user_id = self.clients[tid].id  # Get the user's ID

        if self.cr.user_exists(user_id):  # Check if user exists
            self.cr.delete_user(user_id)  # Delete user from the database
            self.clients[tid].id = tid  # Reset client ID
            self.clients[tid].user = "guest"  # Mark client as logged out
            reply = f"DELR|{email}"  # Confirm deletion to client
        else:
            reply = Errors.LOGIN_DETAILS.value  # Return error if user does not exist

    elif code == "FILS" or code == "UPFL": # Client requests to start uploading a file
        file_name, parent, size, file_id = fields[1:5]
        size = int(size)  # Convert size to integer

        try:
            if self.is_guest(tid):  # Ensure the user is logged in
                reply = Errors.NOT_LOGGED.value
            elif not self.cr.is_dir_owner(self.clients[tid].id, parent):  # Check directory ownership
                reply = Errors.NO_PERMS.value
            elif size > Limits(self.clients[tid].subscription_level).max_file_size * 1_000_000:
                reply = Errors.SIZE_LIMIT.value + f" {Limits(self.clients[tid].subscription_level).max_file_size} MB"
            elif self.cr.get_user_storage(self.clients[tid].user) >
Limits(self.clients[tid].subscription_level).max_storage * 1_000_000:
                reply = Errors.MAX_STORAGE.value
            elif file_id in self.files_uploading.keys():
                reply = Errors.ALREADY_UPLOADING.value
            else:
                # Generate a new file record
                if code == "UPFL":
                    name = self.cr.get_file_sname(file_name)
                    if os.path.exists(CLOUD_PATH + "\\" + name):
                        os.remove(CLOUD_PATH + "\\" + name)  # Delete existing file
                    self.files_uploading[file_id] = File(name, parent, size, file_id, file_name)
                    self.cr.update_file_size(file_name, size)  # Update size in the database
                    reply = f"UPFR|{file_name}|was updated successfully"
                else:
                    name = self.cr.gen_file_name()  # Generate a unique name
                    self.files_uploading[file_id] = File(name, parent, size, file_id, file_name)
                    reply = f"FISS|{file_name}|Upload started"
        except Exception:
            print(traceback.format_exc())  # Log any errors
            reply = Errors.FILE_UPLOAD.value  # Return upload error

    elif code == "FILD" or code == "FILE": # File chunk received from client
        file_id = fields[1].decode()
        location_infile = int(fields[2].decode())
        data = request[4 + len(file_id) + len(str(location_infile)) + 3:]  # Extract file data

        file = None
        for i in range(5):  # Retry logic to ensure file is in tracking list
            if file_id in self.files_uploading.keys():
                file = self.files_uploading[file_id]
                break
            time.sleep(1)

        if file is None:
            return Errors.FILE_NOT_FOUND.value + "|" + file_id  # Return error if file is missing

        # Permission and storage checks
        if self.is_guest(tid):
            reply = Errors.NOT_LOGGED.value
        elif not self.cr.is_dir_owner(self.clients[tid].id, file.parent):
            reply = Errors.NO_PERMS.value
        elif file.size > Limits(self.clients[tid].subscription_level).max_file_size * 1_000_000:
            reply = Errors.SIZE_LIMIT.value + f" {Limits(self.clients[tid].subscription_level).max_file_size} MB"
        elif self.cr.get_user_storage(self.clients[tid].user) >
Limits(self.clients[tid].subscription_level).max_storage * 1_000_000:
            reply = Errors.MAX_STORAGE.value
        else:
            if location_infile + len(data) > file.size:
                return Errors.FILE_SIZE.value  # Ensure data does not exceed allocated size
            file.add_data(data, location_infile)  # Write data to the file

            if code == "FILE":  # Final chunk received
                if file.name != self.clients[tid].user:
                    self.cr.new_file(file.name, file.file_name, file.parent, self.clients[tid].id, file.size)
```

```
                    reply = f"FILR|{file.file_name}|File finished uploading"
                else:
                    reply = f"ICUP|Profile icon uploaded"

                if file_id in self.files_uploading.keys():
                    del self.files_uploading[file_id]  # Remove from tracking list
            else:
                reply = ""  # Continue uploading


        elif (code == "GETP" or code == "GETD" or code == "GESP" or code == "GESD" or code == "GEDP" or code == "GEDD"): #
Client requests files or directories list (personal/shared/deleted)
            directory = fields[1]  # Directory ID or path
            amount = int(fields[2])  # Number of items to fetch
            sort = fields[3]  # Sorting parameter (Name, Date, Size, etc.)
            sort_direction = fields[4] == "True"  # Sorting order (ascending/descending)

            search_filter = fields[5] if len(fields) == 6 else None  # Optional search filter

            prev_amount = 0  # Keeps track of file count before adding directories

            if (code == "GETP"):  # Get personal files
                items = self.cr.get_files(self.clients[tid].id, directory, search_filter)
                reply = "PATH"
            elif (code == "GETD"):  # Get personal directories
                items = self.cr.get_directories(self.clients[tid].id, directory, search_filter)
                prev_amount = len(self.cr.get_files(self.clients[tid].id, directory, search_filter))  # Track previous
file count
                reply = "PATD"
            elif (code == "GESP"):  # Get shared files
                items = self.cr.get_share_files(self.clients[tid].id, directory, search_filter)
                reply = "PASH"
            elif (code == "GESD"):  # Get shared directories
                items = self.cr.get_share_directories(self.clients[tid].id, directory, search_filter)
                prev_amount = len(self.cr.get_share_files(self.clients[tid].id, directory, search_filter))  # Track
previous file count
                reply = "PASD"
            elif (code == "GEDP"):  # Get deleted files
                items = self.cr.get_deleted_files(self.clients[tid].id, directory, search_filter)
                reply = "PADH"
            elif (code == "GEDD"):  # Get deleted directories
                items = self.cr.get_deleted_directories(self.clients[tid].id, directory, search_filter)
                prev_amount = len(self.cr.get_deleted_files(self.clients[tid].id, directory, search_filter))  # Track
previous file count
                reply = "PADD"

            total = len(items) + prev_amount  # Calculate total items in the directory
            amount -= prev_amount  # Adjust the number of items based on previous count

            if amount > len(items):
                amount = len(items)  # Ensure not exceeding available items
            elif amount < 0:
                amount = 0  # Prevent negative count

            # Sorting logic based on user preference
            if sort == "Name" or ((code == "GETD" or code == "GESD" or code == "GEDD") and sort == "Owner"):
                items = sorted(items, key=lambda x: x.split("~")[0].lower(), reverse=sort_direction)  # Sort by name or
owner

            elif sort == "Date":
                if (code == "GETD" or code == "GESD" or code == "GEDD"):
                    items = sorted(items, key=lambda x: self.cr.str_to_date(x.split("~")[2]), reverse=sort_direction)  #
Sort directories by date
                else:
                    items = sorted(items, key=lambda x: self.cr.str_to_date(x.split("~")[1]), reverse=sort_direction)  #
Sort files by date

            elif sort == "Type" and (code == "GETP" or code == "GESP" or code == "GEDP"):
                items = sorted(items, key=lambda x: x.split("~")[0].split(".")[-1].lower(), reverse=sort_direction)  #
Sort by file extension

            elif sort == "Size":
                if (code == "GETD" or code == "GESD" or code == "GEDD"):
                    items = sorted(items, key=lambda x: int(x.split("~")[3]), reverse=sort_direction)  # Sort directories
by size
                else:
                    items = sorted(items, key=lambda x: int(x.split("~")[2]), reverse=sort_direction)  # Sort files by
size

            elif sort == "Owner" and (code == "GETD" or code == "GESD" or code == "GEDD"):
                items = sorted(items, key=lambda x: x.split("~")[4].lower(), reverse=sort_direction)  # Sort by owner (for
shared content)

            reply += f"|{total}"  # Include total count in response

            for item in items[:amount]:  # Append requested number of items to response
                reply += f"|{item}"
```

```
        elif (code == "MOVD"): # Client requests to move to a different directory
            directory_id = fields[1]  # Extract the target directory ID

            if (self.cr.valid_directory(directory_id, self.clients[tid].id) or directory_id == ""):
                self.clients[tid].cwd = directory_id  # Update current working directory
                reply = f"MOVR|{directory_id}|{self.cr.get_parent_directory(directory_id)}|
{self.cr.get_full_path(directory_id)}|moved successfully"
            else:
                self.clients[tid].cwd = ""  # Reset to root if directory is invalid
                reply = f"MOVR|{''}|{self.cr.get_parent_directory('')}|{self.cr.get_full_path('')}|moved successfully"

        elif (code == "DOWN"): # Client requests to download a file or folder
            file_id = fields[1]

            if "~" in file_id:  # Multiple files requested (zip them)
                name = fields[2]  # Name of the zip file
                ids = file_id.split("~")  # Split multiple file IDs

                for id in ids:
                    if not self.cr.can_download(self.clients[tid].id, id) or self.is_guest(tid):
                        reply = Errors.NO_PERMS.value  # Permission denied
                        return reply
                    elif self.cr.get_file_sname(id) is None and self.cr.get_dir_name(id) is None:
                        reply = Errors.FILE_NOT_FOUND.value + "|" + file_id  # File not found
                        return reply

                zip_buffer = self.cr.zip_files(ids)  # Create a zip archive of the files
                self.send_zip(zip_buffer, file_id, sock, tid)  # Send the zipped file to the client
                zip_buffer.close()
                reply = f"DOWR|{name}|{file_id}|was downloaded"
            else:
                if not self.cr.can_download(self.clients[tid].id, file_id) or self.is_guest(tid):
                    reply = Errors.NO_PERMS.value  # Permission denied
                    return reply
                elif self.cr.get_dir_name(file_id) is not None:
                    zip_buffer = self.cr.zip_directory(file_id)  # Zip the entire directory
                    self.send_zip(zip_buffer, file_id, sock, tid)  # Send the zip file
                    zip_buffer.close()
                    reply = f"DOWR|{self.cr.get_dir_name(file_id)}|{file_id}|was downloaded"
                    return reply
                elif self.cr.get_file_sname(file_id) is None:
                    reply = Errors.FILE_NOT_FOUND.value + "|" + file_id  # File not found
                    return reply

                file_path = CLOUD_PATH + "\\" + self.cr.get_file_sname(file_id)  # Get the full file path
                if not os.path.isfile(file_path):
                    reply = Errors.FILE_NOT_FOUND.value + "|" + file_id  # Ensure the file exists
                else:
                    try:
                        self.send_file_data(file_path, file_id, sock, tid)  # Send file data
                        reply = f"DOWR|{self.cr.get_file_fname(file_id)}|was downloaded"
                    except Exception:
                        reply = Errors.FILE_DOWNLOAD.value  # Handle file download errors

        elif (code == "NEWF"): # Client requests to create a new folder
            folder_name = fields[1]
            folder_path = self.clients[tid].cwd  # Get current directory

            if not self.cr.is_dir_owner(self.clients[tid].id, folder_path) or self.is_guest(tid):
                reply = Errors.NO_PERMS.value  # Permission denied
            else:
                self.cr.create_folder(folder_name, folder_path, self.clients[tid].id)  # Create the folder
                reply = f"NEFR|{folder_name}|was created"

        elif (code == "RENA"): # Client requests to rename a file or directory
            file_id, name, new_name = fields[1:4]

            if self.v.is_empty(fields[1:]):
                return Errors.EMPTY_FIELD.value  # Ensure fields are not empty
            elif not self.cr.can_rename(self.clients[tid].id, file_id):
                reply = Errors.NO_PERMS.value  # Permission denied
            else:
                if self.cr.get_file_fname(file_id) is not None:
                    self.cr.rename_file(file_id, new_name)  # Rename file
                else:
                    self.cr.rename_directory(file_id, new_name)  # Rename directory
                reply = f"RENR|{name}|{new_name}|File renamed successfully"

        elif (code == "GICO"): # sending user icon
            if (os.path.isfile(os.path.join(USER_ICONS_PATH, self.clients[tid].id) + ".ico")): # checl of user has icon
                self.send_file_data(os.path.join(USER_ICONS_PATH, self.clients[tid].id) + ".ico", "user", sock, tid)
            else:
                self.send_file_data(os.path.join(USER_ICONS_PATH, "guest.ico"), "user", sock, tid) # send generic icon
            reply = f"GICR|Sent use profile picture"

        elif (code == "ICOS"): # uploading new user icon
```

```
        size = int(fields[3])
        id = fields[4]
        try:
            self.files_uploading[id] = File(self.clients[tid].id, "", size, id, self.clients[tid].id, icon=True)
            reply = f"ICOR|Profile icon started uploading"

        except Exception:
            print(traceback.format_exc())
            reply = Errors.FILE_UPLOAD.value

    elif (code == "DELF"): # Client requests to delete a file or folder
        file_id = fields[1]

        if not self.cr.can_delete(self.clients[tid].id, file_id):
            reply = Errors.NO_PERMS.value  # Permission denied
        elif file_id in self.files_in_use:
            reply = Errors.IN_USE.value  # File is currently in use
        elif self.cr.get_file_fname(file_id) is not None:
            name = self.cr.get_file_fname(file_id)
            self.cr.delete_file(file_id)  # Delete file from storage
            reply = f"DLFR|{name}|was deleted!"
        elif self.cr.get_dir_name(file_id) is not None:
            name = self.cr.get_dir_name(file_id)
            self.cr.delete_directory(file_id)  # Delete directory
            reply = f"DFFR|{name}|was deleted!"
        else:
            reply = Errors.FILE_NOT_FOUND.value + "|" + file_id  # File not found


    elif code == "SUBL": # change subscription level
        level = fields[1]
        if (level == self.clients[tid].subscription_level): # check level validity
            reply = Errors.SAME_LEVEL.value
        elif (int(level) < 0 or int(level) > 3):
            reply = Errors.INVALID_LEVEL.value
        else:
            self.cr.change_level(self.clients[tid].id, int(level)) # change it
            self.clients[tid].subscription_level = int(level)
            reply = f"SUBR|{level}|Subscription level updated"

    elif (code == "GEUS"): # Client requests current storage usage
        used_storage = self.cr.get_user_storage(self.clients[tid].id)  # Get userâ€™s used storage
        reply = f"GEUR|{used_storage}"

    elif (code == "CHUN"): # Client requests to change their username
        new_username = fields[1]

        if self.v.is_empty(fields[1:]):
            return Errors.EMPTY_FIELD.value  # Ensure field is not empty
        elif self.v.check_illegal_chars(fields[1:]):
            return Errors.INVALID_CHARS.value  # Check for illegal characters
        elif not self.v.is_valid_username(new_username) or new_username == "guest":
            return Errors.INVALID_USERNAME.value  # Validate username
        elif self.cr.user_exists(new_username):
            reply = Errors.USER_REGISTERED.value  # Username is already taken
        else:
            self.cr.change_username(self.clients[tid].id, new_username)  # Update username in database
            self.clients[tid].user = new_username  # Update username in session
            reply = f"CHUR|{new_username}|Changed username"

    elif code == "VIEW": # send file to view
        file_id = fields[1]
        file_path = CLOUD_PATH + "\\" + self.cr.get_file_sname(file_id)
        if(not self.cr.can_download(self.clients[tid].id, file_id)):
            reply = Errors.NO_PERMS.value + "|" + self.cr.get_file_fname(file_id)
        elif (not os.path.isfile(file_path)):
            reply = Errors.FILE_NOT_FOUND.value + "|" + file_id
        elif (os.path.getsize(file_path) > 10_000_000):
            reply = f"{Errors.PREVIEW_SIZE.value}|{self.cr.get_file_fname(file_id)}"
        elif file_id in self.files_in_use: # if file is already in view dont allow
            reply = Errors.IN_USE.value
        else:
            try:
                self.send_file_data(file_path, file_id, sock, tid) # send the file
                self.files_in_use.append(file_id)
                reply = f"VIER|{self.cr.get_file_fname(file_id)}|was viewed"
            except Exception:
                reply = Errors.FILE_DOWNLOAD.value


    elif (code == "GENC"): # Client requests to generate a new authentication cookie
        if self.is_guest(tid):
            reply = Errors.NOT_LOGGED.value  # Ensure user is logged in
        else:
            self.cr.generate_cookie(self.clients[tid].id)  # Generate new cookie
            reply = f"COOK|{self.cr.get_cookie(self.clients[tid].id)}"
```

```
        elif (code == "COKE"): # Client sends authentication cookie for validation
            cookie = fields[1]
            user_dict = self.cr.get_user_data(cookie)  # Retrieve user data

            if user_dict is None:
                reply = Errors.INVALID_COOKIE.value  # Invalid cookie
            elif self.cr.cookie_expired(user_dict["id"]):
                reply = Errors.EXPIRED_COOKIE.value  # Expired cookie
            else:
                username = user_dict["username"]
                email = user_dict["email"]
                self.clients[tid].id = user_dict["id"]
                self.clients[tid].user = user_dict["username"]
                self.clients[tid].email = user_dict["email"]
                self.clients[tid].subscription_level = user_dict["subscription_level"]
                self.clients[tid].admin_level = user_dict["admin_level"]
                reply = f"LOGS|{email}|{username}|{int(self.clients[tid].subscription_level)}|
{self.clients[tid].admin_level}"

        elif (code == "SHRS"): # Client requests to share a file or folder with another user
            file_id = fields[1]  # File or folder ID
            user_cred = fields[2]  # Email or username of the recipient

            if self.cr.get_file_fname(file_id) is None and self.cr.get_dir_name(file_id) is None:
                reply = Errors.FILE_NOT_FOUND.value + "|" + file_id  # File or folder does not exist
            elif not self.cr.can_share(self.clients[tid].id, file_id):
                reply = Errors.NO_PERMS.value  # User does not have permission to share this file
            elif user_cred == self.clients[tid].email or user_cred == self.clients[tid].user:
                reply = Errors.SELF_SHARE.value  # Cannot share a file with yourself
            elif self.cr.is_file_owner(self.cr.get_user_id(user_cred), file_id) or
self.cr.is_dir_owner(self.cr.get_user_id(user_cred), file_id):
                reply = Errors.OWNER_SHARE.value  # Cannot share a file with its owner
            elif self.cr.get_user_data(user_cred) is None:
                reply = Errors.USER_NOT_FOUND.value  # Recipient user not found
            else:
                sharing = self.cr.get_share_options(file_id, user_cred)  # Get existing share settings
                if sharing is None:
                    reply = f"SHRR|{file_id}|{user_cred}|{self.cr.get_file_fname(file_id)}"  # File is not yet shared
                else:
                    reply = f"SHRR|{file_id}|{user_cred}|{self.cr.get_file_fname(file_id)}|" + "|".join(sharing[4:])  #
Return existing share settings

        elif (code == "SHRP"): # Client updates sharing permissions for a file or folder
            file_id = fields[1]  # File ID
            user_cred = fields[2]  # Email or username of recipient

            if self.cr.get_file_fname(file_id) is None and self.cr.get_dir_name(file_id) is None:
                reply = Errors.FILE_NOT_FOUND.value + "|" + file_id  # File does not exist
            elif not self.cr.can_share(self.clients[tid].id, file_id):
                reply = Errors.NO_PERMS.value  # User lacks permission to share
            elif user_cred == self.clients[tid].email or user_cred == self.clients[tid].user:
                reply = Errors.SELF_SHARE.value  # Cannot share with yourself
            elif self.cr.is_file_owner(self.cr.get_user_id(user_cred), file_id) or
self.cr.is_dir_owner(self.cr.get_user_id(user_cred), file_id):
                reply = Errors.OWNER_SHARE.value  # Cannot share with the owner
            elif self.cr.get_user_data(user_cred) is None:
                reply = Errors.USER_NOT_FOUND.value  # Recipient not found
            else:
                self.cr.share_file(file_id, user_cred, fields[3:])  # Update sharing permissions
                reply = f"SHPR|Sharing option with {user_cred} have been updated"

        elif code == "SHRE": # Client requests to remove sharing permissions for a file or folder
            file_id = fields[1]  # File ID
            file_name = self.cr.get_file_fname(file_id) or self.cr.get_dir_name(file_id)  # Get file or folder name

            if file_name is None:
                reply = Errors.FILE_NOT_FOUND.value + "|" + file_id  # File not found
            else:
                self.cr.remove_share(self.clients[tid].id, file_id)  # Remove sharing permissions
                reply = f"SHRM|{file_name}|Share removed"

        elif code == "RECO": # Client requests to recover a deleted file or folder
            file_id = fields[1]

            if not self.cr.can_delete(self.clients[tid].id, file_id):
                reply = Errors.NO_PERMS.value  # User lacks permission to recover
            elif self.cr.get_file_fname(file_id) is not None:
                name = self.cr.get_file_fname(file_id)  # Get file name
            elif self.cr.get_dir_name(file_id) is not None:
                name = self.cr.get_dir_name(file_id)  # Get folder name

            if name is None:
                reply = Errors.FILE_NOT_FOUND.value + "|" + file_id  # File not found
            else:
                self.cr.recover(file_id)  # Restore file from deleted state
                reply = f"RECR|{name}|was recovered!"
```

```
        elif code == "VIEE": # Client stops viewing a file
            file_id = fields[1]
            self.files_in_use.remove(file_id)  # Remove file from in-use list
            reply = f"VIRR|{file_id}|stop viewing"

        elif code == "STOP": # Client requests to stop an ongoing file upload
            file_id = fields[1]
            name = self.remove_file_mid_down(file_id)  # Remove file from upload list
            reply = f"STOR|{name}|{file_id}|File upload stopped"

        elif code == "RESU": # Client requests to resume an interrupted file upload
            file_id = fields[1]

            if file_id in self.files_uploading.keys():
                progress = self.files_uploading[file_id].curr_location_infile  # Get current upload progress
                reply = f"RESR|{file_id}|{progress}"  # Send resume position
            else:
                reply = Errors.FILE_NOT_FOUND.value + "|" + file_id  # File not found

        elif code == "RESD": # Client requests to resume an interrupted file download
            id = fields[1]
            progress = int(fields[2])
            if self.cr.get_file_sname(id) != None:
                file_path = CLOUD_PATH + "\\" + self.cr.get_file_sname(id)
                self.send_file_data(file_path, id, sock, tid, progress) # if file send file
            elif self.cr.get_dir_name(id) != None:
                zip_buffer = self.cr.zip_directory(id) # if folder send zip
                self.send_zip(zip_buffer, id, sock, tid)
                zip_buffer.close()
            elif "~" in id:
                ids = id.split("~")
                zip_buffer = self.cr.zip_files(ids)
                self.send_zip(zip_buffer, id, sock, tid)
                zip_buffer.close()
            else:
                reply = Errors.FILE_NOT_FOUND.value + "|" + id
                return reply
            reply = f"RUSR|{id}|{progress}"

        elif code == "UPDT": # Client sends an update message to the server
            msg = fields[1]  # Message content
            reply = f"UPDR|{msg}"  # Send update message back

        elif code == "ADMN": # Client requests admin data
            if self.clients[tid].admin_level > 0:
                reply = f"ADMR"
                for user in self.cr.get_admin_table():
                    id, email, username, verified, subscription_level, admin_level = user[0], user[1], user[2], user[7],
user[8], user[9]
                    files_amount, total_storage_used = self.cr.get_user_total_files(id), self.cr.get_user_storage(id)
                    reply +=f"|{id}~{email}~{username}~{verified}~{subscription_level}~{admin_level}~{files_amount}~
{total_storage_used}"

            else:
                reply = Errors.NO_PERMS.value
        else:
            # Unknown request received
            reply = Errors.UNKNOWN.value  # Return generic error message
            fields = ''

        return reply  # Send response to client

    def send_file_data(self, file_path, id, sock, tid, progress=0):
        """
        Sends a file's content to the client in chunks.
        Supports resuming from a given progress point.
        """
        lock_path = f"{file_path}.lock"  # Lock file path to prevent concurrent access
        lock = FileLock(lock_path)  # Create a lock object

        if not os.path.isfile(file_path):
            raise Exception  # Raise an error if file doesn't exist

        size = os.path.getsize(file_path)  # Get total file size
        left = size % CHUNK_SIZE  # Get remainder bytes outside full chunks
        sent = progress  # Track amount of data sent

        start = time.time()
        bytes_sent = 0

        try:
            with lock:  # Acquire lock before reading the file
                with open(file_path, 'rb') as f:
                    f.seek(progress)  # Move file pointer to resume position

                    for i in range((size - progress) // CHUNK_SIZE):
                        location_infile = f.tell()  # Get current position
```

```
                    data = f.read(CHUNK_SIZE)  # Read file in chunks
                    current_time = time.time()
                    elapsed_time = current_time - start

                    if elapsed_time >= 1.0:  # Reset bandwidth tracking every second
                        start = current_time
                        bytes_sent = 0

                    self.network.send_data(sock, tid, f"RILD|{id}|{location_infile}|".encode() + data)
                    bytes_sent += len(data)  # Update sent bytes counter
                    sent += CHUNK_SIZE

                    # Check if bandwidth limit is exceeded
                    if bytes_sent >= (Limits(self.clients[tid].subscription_level).max_download_speed) * 1_000_000:
                        time_to_wait = 1.0 - elapsed_time
                        if time_to_wait > 0:
                            time.sleep(time_to_wait)  # Pause to respect speed limit

                location_infile = f.tell()
                data = f.read(left)  # Read remaining bytes
                if data != b"":
                    self.network.send_data(sock, tid, f"RILE|{id}|{location_infile}|".encode() + data)  # Send last
chunk
        except:
            print(traceback.format_exc())  # Log error
            if os.path.exists(lock_path):
                os.remove(lock_path)  # Remove lock file if an error occurs
            raise

    def send_zip(self, zip_buffer, id, sock, tid, progress=0):
        """
        Sends a compressed zip file to the client in chunks.
        Supports resuming from a given progress point.
        """
        size = len(zip_buffer.getbuffer())  # Get zip file size
        left = size % CHUNK_SIZE  # Get remainder bytes
        sent = progress  # Track amount sent

        start = time.time()
        bytes_sent = 0

        try:
            zip_buffer.seek(progress)  # Move to the resume position

            for i in range((size - progress) // CHUNK_SIZE):
                location_infile = zip_buffer.tell()
                data = zip_buffer.read(CHUNK_SIZE)  # Read zip file in chunks
                current_time = time.time()
                elapsed_time = current_time - start

                if elapsed_time >= 1.0:  # Reset tracking every second
                    start = current_time
                    bytes_sent = 0

                self.network.send_data(sock, tid, f"RILD|{id}|{location_infile}|".encode() + data)
                bytes_sent += len(data)
                sent += CHUNK_SIZE

                # Check if bandwidth limit is exceeded
                if bytes_sent >= (Limits(self.clients[tid].subscription_level).max_download_speed) * 1_000_000:
                    time_to_wait = 1.0 - elapsed_time
                    if time_to_wait > 0:
                        time.sleep(time_to_wait)  # Pause to respect speed limit

            location_infile = zip_buffer.tell()
            data = zip_buffer.read(left)  # Read last part
            if data != b"":
                self.network.send_data(sock, tid, f"RILE|{id}|{location_infile}|".encode() + data)  # Send final chunk
        except:
            raise  # Raise any encountered exception

    def is_guest(self, tid):
        """
        Checks if a client is a guest user.
        """
        return self.clients[tid].user == "guest"

    def remove_file_mid_down(self, id):
        """
        Cancels an ongoing file upload and deletes the file.
        """
        if id in self.files_uploading.keys():
            name = self.files_uploading[id].file_name
            file_id = self.cr.get_file_id(self.files_uploading[id].name)  # Retrieve actual file ID
            self.cr.delete_file(file_id)  # Delete file from database
            del self.files_uploading[id]  # Remove file from active uploads
            return name  # Return the name of the deleted file
```

```
class File:
    """
    Represents a file being uploaded or downloaded.
    Handles file storage, tracking, and writing data incrementally.
    """

    def __init__(self, name, parent, size, id, file_name, curr_location_infile=0, icon=False):
        self.name = name  # Internal storage name of the file
        self.parent = parent  # Parent directory ID
        self.size = size  # File size in bytes
        self.id = id  # Unique file identifier
        self.file_name = file_name  # Original file name as uploaded
        self.curr_location_infile = curr_location_infile  # Tracks upload/download progress
        self.save_path = USER_ICONS_PATH + "\\" + self.name + ".ico" if icon else CLOUD_PATH + "\\" + self.name  #
Determine save location
        self.start_download()  # Initialize the file for writing

    def start_download(self):
        """
        Prepares the file for writing by creating an empty file of the correct size.
        """
        with open(self.save_path, 'wb') as f:
            f.seek(self.size - 1)  # Move to the last byte of the file
            f.write(b"\0")  # Write a null byte to allocate space
            f.flush()  # Ensure data is written to disk

    def add_data(self, data, location_infile):
        """
        Writes received data to the file at the specified location.
        Uses file locking to prevent simultaneous writes.
        """
        lock_path = f"{self.save_path}.lock"  # Define a lock file path
        lock = FileLock(lock_path)  # Create a lock object

        try:
            with lock:  # Acquire lock before writing
                with open(self.save_path, 'r+b') as f:
                    f.seek(location_infile)  # Move to the correct write position
                    f.write(data)  # Write received data
                    f.flush()  # Ensure data is written to disk
                    self.curr_location_infile = location_infile  # Update progress tracking
        except:
            print(traceback.format_exc())  # Print error if write operation fails
            self.uploading = False  # Mark file as inactive due to failure
        finally:
            try:
                if os.path.exists(lock_path):  # Remove lock file if it exists
                    os.remove(lock_path)
            except:
                pass  # Ignore errors when removing the lock file

    def delete(self):
        """
        Deletes the file from storage.
        Ensures any lock file is removed before deleting the actual file.
        """
        lock_path = f"{self.save_path}.lock"
        if os.path.exists(lock_path):
            os.remove(lock_path)  # Remove lock file if it exists
        if os.path.exists(self.save_path):
            os.remove(self.save_path)  # Delete the actual file
```