Dropbox-like File Sharing Simulator: Technical Report

1. Introduction

This document provides a detailed technical analysis of the Dropbox-like file sharing simulator implemented in Python. The system utilizes a client-server architecture built upon fundamental socket programming principles, deliberately avoiding higher-level networking libraries to fulfill specific requirements.

The primary goal is to simulate core file synchronization functionalities, enabling multiple clients connected to a central server to maintain consistent views of a shared folder. Key features include the creation, deletion, and updating of both files (including binary content) and folders. Synchronization is achieved through a combination of real-time broadcasts triggered by client actions and periodic polling by clients to fetch missed updates.

This report delves into the design choices, implementation details, and communication protocols employed in the system. It examines the server's role in managing the central repository and client connections, the client's responsibilities in monitoring local changes and interacting with the server, and the underlying protocol that facilitates their communication.

Simplifications: For the purpose of this simulation, several aspects common in production systems are intentionally omitted:

- **User Authentication:** No user accounts or login mechanisms are implemented.
- **Security:** Communication is not encrypted, and there are no access control measures.
- **Graphical User Interface (GUI):** Interaction is purely command-line based.
- **Conflict Resolution:** The system assumes a simple "last write wins" strategy without sophisticated conflict detection or resolution logic.

- **Advanced Features:** Features like version history, selective sync, or detailed permission management are not included.

2. System Architecture

The simulator adopts a classic client-server model.

2.1. Server (`server.py`, `main_server.py`)

- **Role:** Acts as the central authority, managing the definitive state of the shared folder and coordinating communication between clients.
- **Core Components:**
    - `FileServer`: The main class orchestrating server operations. It binds to a specified host and port, listens for incoming client connections, and manages a pool of active client handlers.
    - `ClientHandler`: A dedicated thread for each connected client. It handles all communication with that specific client, processing incoming requests and sending necessary updates or acknowledgments.
    - `FileChangeTracker`: Responsible for maintaining the state of the shared folder on the server side. It tracks file/folder modification timestamps and keeps a record of deleted items to efficiently determine what updates need to be sent to clients during polling.
- **Functionality:**
    - **Connection Management:** Accepts new TCP connections from clients and spawns a `ClientHandler` thread for each. Manages the lifecycle of these connections, removing handlers when clients disconnect.
    - **Shared Folder Management:** Maintains a designated directory on the server's filesystem (`server_shared_folder` by default). All synchronized files and folders reside here.
    - **Request Handling:** Processes various actions requested by clients (create, update, delete files/folders) via the `ClientHandler`.
    - **State Tracking:** Uses `FileChangeTracker` to keep track of the latest modification times for all items in the shared folder and remembers which items have been deleted.

    - **Synchronization Logic:**
        - **Broadcasting:** When a client action modifies the shared folder (e.g., creating a file), the server immediately broadcasts this change to *all other* connected clients to ensure near real-time updates.
        - **Polling Support:** Responds to client requests (`ACTION_REQUEST_UPDATES`) by sending a list of items changed or deleted since the client's last reported synchronization time.
        - **Initial Sync:** Handles requests (`ACTION_REQUEST_ALL_FILES`) from newly connected clients, sending the complete current state (all files and folders with content) of the shared folder.
- **Concurrency:** Employs multi-threading, with each client handled by a separate `ClientHandler` thread. Access to the shared list of clients (`FileServer.clients`) is protected by a `threading.Lock` (`clients_lock`) to prevent race conditions during connection/disconnection and broadcasting.

2.2. Client (`client.py`, `main_client.py`)

- **Role:** Represents an endpoint user device. It maintains a local replica of the shared folder and synchronizes it with the server.
- **Core Components:**
    - `FileClient`: The main class managing client operations. It connects to the server, initializes synchronization, monitors the local folder for changes, and handles communication with the server.
    - `FileSystemChangeHandler`: A `watchdog` event handler that detects local filesystem modifications (creation, deletion, modification, moves) within the client's designated folder.
    - `Observer`: The `watchdog` observer thread that monitors the local folder using the `FileSystemChangeHandler`.
- **Functionality:**
    - **Server Connection:** Establishes and maintains a TCP connection with the `FileServer`. Includes logic for attempting reconnection if the connection is lost.

    - **Local Folder Management:** Manages a local directory (`client_folder` by default) intended to mirror the server's shared folder.
    - **Local Change Detection:** Uses the `watchdog` library to monitor the local folder for any changes made by the user (or other processes). It filters out common temporary files.
    - **Sending Local Changes:** When a local change is detected, the client sends a corresponding message (e.g., `ACTION_CREATE_FILE`, `ACTION_DELETE_FOLDER`) to the server.
    - **Receiving Remote Changes:**
        - **Broadcast Handling:** Listens for broadcast messages from the server (e.g., `ACTION_UPDATE_FILE`, `ACTION_DELETE_FILE`) indicating changes made by other clients. It applies these changes directly to the local folder.
        - **Polling:** Periodically (configurable interval) sends `ACTION_REQUEST_UPDATES` to the server with its `last_sync_time` to fetch any changes missed between polls or broadcasts. It processes the `ACTION_SEND_UPDATES` response from the server.
        - **Initial Sync:** Upon connecting, sends `ACTION_REQUEST_ALL_FILES` to get the server's complete current state and populates its local folder accordingly.
    - **Applying Changes:** Modifies its local filesystem (creating files/folders, writing data, deleting items) based on instructions received from the server. It uses an `ignore_next_event_for` mechanism in the `FileSystemChangeHandler` to prevent reacting to filesystem events that the client itself caused while applying server updates.
- **Concurrency:** Runs the server communication (polling and receiving broadcasts) and local filesystem monitoring (`watchdog`) in separate threads (`poll_thread` and the `Observer` thread, respectively). A `threading.Lock` (`sync_lock`) is used to synchronize access to the shared socket resource and potentially related state when handling local changes or processing server messages.

2.3. Communication Protocol (`protocol.py`)

- **Role:** Defines the structure and rules for messages exchanged between the server and clients. Ensures both parties can understand each other.
- **Format:** Uses JSON for message payloads, making them human-readable and easy to parse. File content is Base64 encoded to allow binary data transmission within the JSON structure.

- **Structure:** Each message consists of:
    - **Header:** A fixed-size (8 bytes) header indicating the total length of the subsequent JSON payload. This allows the receiver to know exactly how many bytes to read for the complete message.
    - **Payload:** A JSON object containing:
        - `action`: A string specifying the purpose of the message (e.g., `ACTION_CREATE_FILE`, `ACTION_REQUEST_UPDATES`).
        - `payload`: A nested JSON object containing the data relevant to the action (e.g., file path, Base64 encoded data, timestamp, list of updates).
- **Key Actions:**
    - `CONNECT/DISCONNECT`: Manage client connection lifecycle.
    - `CREATE_FILE/UPDATE_FILE/DELETE_FILE`: Client requests to modify files; Server broadcasts of file changes.
    - `CREATE_FOLDER/DELETE_FOLDER`: Client requests to modify folders; Server broadcasts of folder changes.
    - `REQUEST_UPDATES`: Client polls for changes since `last_sync`.
    - `SEND_UPDATES`: Server response to polling, listing updated/deleted items.
    - `REQUEST_ALL_FILES`: Client request for initial full state.
    - `SEND_ALL_FILES`: Server response with the complete list of files and folders.
    - `ACK`: Generic acknowledgment from server to client, usually indicating successful processing of a request.
    - `ERROR`: Indicates an error occurred during processing on the server.
- **Serialization:** Provides helper functions:
    - `encode_file_data`: Reads a file's content and encodes it into a Base64 string. Returns `None` for empty files.
    - `decode_file_data`: Decodes a Base64 string (or handles `None` for empty files) and writes the resulting bytes to a specified file path, creating parent directories if necessary.
- **Transmission:** Includes functions for reliable sending (`send_message`) and receiving (`receive_message`) of these length-prefixed messages over TCP sockets, handling potential partial reads/writes. Defines `MAX_MESSAGE_SIZE` to prevent excessively large messages.

3. Detailed Implementation Analysis

3.1. Server Implementation (`server.py`)

3.1.1. `FileServer` Class

- **Initialization (`__init__`)**:
    - Stores host, port, and the absolute path to the shared folder.
    - Ensures the shared folder directory exists using `os.makedirs(exist_ok=True)`.
    - Creates the main TCP server socket (`socket.AF_INET`, `socket.SOCK_STREAM`).
    - Sets the `SO_REUSEADDR` socket option to allow the server to restart quickly without waiting for the OS to release the port.
    - Instantiates the `FileChangeTracker`.
    - Initializes an empty list (`clients`) to hold `ClientHandler` instances and a `threading.Lock` (`clients_lock`) to protect it.
    - Sets the initial `running` flag to `False`.
- **Starting (`start`)**:
    - Binds the server socket to the specified host and port.
    - Puts the socket into listening mode (`listen(8)` allows a backlog of 8 pending connections).
    - Logs server start information.
    - Sets the `running` flag to `True`.
    - Enters the main loop (`while self.running`):
        - Blocks on `self.server_socket.accept()`, waiting for a new client connection.
        - Upon connection, receives the client socket and address.
        - Calls `_handle_new_client` to process the new connection.
        - Includes basic error handling for `accept()` failures (e.g., if the socket is closed while accepting).

    - Uses a `try...finally` block to ensure `stop()` is called even if errors occur during startup or the main loop.
- **Stopping (`stop`)**:
    - Logs the shutdown sequence.
    - Sets the `running` flag to `False` to break the main accept loop.
    - Acquires the `clients_lock`.
    - Iterates through a copy (`self.clients[:]`) of the client list:
        - Sets the client handler's `running` flag to `False`.
        - Attempts to close the client's socket (`client.socket.close()`). Includes basic error handling for the close operation.
    - Clears the `clients` list.
    - Releases the `clients_lock`.
    - Attempts to close the main server socket (`self.server_socket.close()`).
    - Logs server stop.
- **Handling New Clients (`_handle_new_client`)**:
    - Creates a `ClientHandler` instance, passing the client socket, address, and the `FileServer` instance itself.
    - Acquires the `clients_lock`.
    - Appends the new `ClientHandler` to the `clients` list.
    - Releases the `clients_lock`.
    - Starts the `ClientHandler` thread (`client_handler.start()`).
- **Removing Clients (`remove_client`)**:
    - Acquires the `clients_lock`.
    - Safely removes the specified `ClientHandler` instance from the `clients` list if it exists.
    - Releases the `clients_lock`. This is called by `ClientHandler` upon disconnection or error.
- **Broadcasting (`broadcast_file_update`, `broadcast_file_deletion`, etc.)**:
    - These methods are used to notify other clients about changes initiated by one client.

    - Acquires the `clients_lock`.
    - Iterates through the `clients` list.
    - Skips the `exclude_client` (the one that initiated the change).
    - For each other client:
        - Creates the appropriate message using `protocol` helper functions (e.g., `protocol.create_update_file_message`).
        - Attempts to send the message using `protocol.send_message`.
        - Logs the broadcast action or any errors encountered during sending.
    - Releases the `clients_lock`.

3.1.2. `ClientHandler` Class (Thread)

- **Initialization (`__init__`)**:
    - Stores the client socket, address, and the parent `FileServer` instance.
    - Initializes the `running` flag to `True`.
    - Calls the `threading.Thread` superclass initializer.
- **Main Loop (`run`)**:
    - Logs client connection.
    - Enters the main loop (`while self.running`):
        - Calls `protocol.receive_message` to wait for and receive a complete message from the client socket. This handles the header parsing and message reconstruction.
        - Calls `_handle_message` to process the received message.
        - Includes error handling for `ConnectionError` or `ValueError` during message reception, breaking the loop if errors occur.
    - Uses a `try...except...finally` structure:
        - Catches general exceptions during message handling.

        - The `finally` block ensures that client disconnection is logged, the socket is closed (`self.socket.close()`), and the handler removes itself from the server's list (`self.server.remove_client(self)`).

- **Message Handling (`_handle_message`)**:
    - Extracts the `action` and `payload` from the received message dictionary.
    - Uses an `if/elif/else` structure to delegate processing based on the `action` string:
        - Calls specific private methods (`_handle_connect`, `_handle_create_file`, etc.) for known actions.
        - Logs a warning and sends an error message back to the client for unknown actions.
    - Includes a broad `try...except` block around the action handling to catch errors during processing and send an error message back to the client.
- **Action Handlers (`_handle_create_file`, `_handle_update_file`, etc.)**:
    - Each handler corresponds to a specific protocol action initiated by the client.
    - **Payload Validation:** They typically start by extracting necessary fields (`path`, `data`, `timestamp`, `last_sync`) from the `payload` dictionary and perform basic validation (checking existence and sometimes type). If validation fails, `_send_error` is called, and the handler returns.
    - **Path Construction:** Constructs the absolute path on the server's filesystem by joining the server's `shared_folder_path` with the relative `path` received from the client.
    - **Filesystem Operations:** Performs the required `os` operations:
        - `CREATE_FILE`/`UPDATE_FILE`: Creates parent directories (`os.makedirs(..., exist_ok=True)`), then uses `protocol.decode_file_data` to write the received (and decoded) data to the `abs_path`. Handles `None` data to create empty files.
        - `DELETE_FILE`: Checks if the `abs_path` exists and is a file (`os.path.exists`, `os.path.isfile`), then calls `os.remove`.
        - `CREATE_FOLDER`: Checks if the `abs_path` doesn't exist (`os.path.exists`), then calls `os.makedirs(..., exist_ok=True)`.

        - `DELETE_FOLDER`: Checks if the `abs_path` exists and is a directory (`os.path.exists`, `os.path.isdir`). If so, it uses `os.walk(..., topdown=False)` to recursively iterate through the folder's contents from the bottom up, removing files (`os.remove`) and then directories (`os.rmdir`). It records *all* deleted relative paths during this process.

    - **Change Tracking:** Calls the appropriate `self.server.change_tracker` method (`record_file_change` or `record_deletion`) to update the server's state view *after* successfully modifying the filesystem. For folder deletion, it calls `record_deletion` for every file and subdirectory within the deleted folder.

    - **Acknowledgment:** Sends an `ACK` message back to the originating client using `_send_ack` upon successful completion.

    - **Broadcasting:** Calls the relevant `self.server.broadcast_...` method to notify *other* clients of the change, passing itself (`self`) as the `exclude_client`. For folder deletion, it broadcasts individual deletion messages for each file and subdirectory that was removed.

    - **Error Handling:** Uses `try...except` blocks around filesystem operations and broadcasting to catch potential errors, log them, and send an `ERROR` message back to the originating client via `_send_error`.

- **Special Handlers:**

    - `_handle_connect`: Simply sends an ACK.

    - `_handle_disconnect`: Sends an ACK and sets `self.running = False` to terminate the handler thread.

    - `_handle_request_updates`: Retrieves the `last_sync` time from the payload. Calls `self.server.change_tracker.get_changes_since()` to get updated and deleted items. Creates a `SEND_UPDATES` message with this data and sends it back to the client.

    - `_handle_request_all_files`: Records the current time (`sync_start_time`). Walks the entire server `shared_folder_path` using `os.walk`. Collects relative paths for all directories and files. For files, it also encodes their content using `protocol.encode_file_data` and gets the modification time (`os.path.getmtime`). Creates a `SEND_ALL_FILES` message containing lists of folders (`folders`) and file details (`files` - path, data, timestamp), along with the `sync_start_time`. Sends this large message back to the client.

- **Sending Messages (`_send_ack`, `_send_error`)**: Helper methods that use `protocol.create_ack_message` or `protocol.create_error_message` and then `protocol.send_message` to send standardized responses to the client.

3.1.3. `FileChangeTracker` Class

- **Initialization (`__init__`)**:
  - Stores the `shared_folder_path`.
  - Initializes `file_timestamps` (a dictionary mapping relative paths to last modification timestamps) and `deleted_items` (a set of relative paths that have been deleted).
  - Calls `_initialize_timestamps` to populate the initial state based on the existing filesystem content.
- **Initial Scan (`_initialize_timestamps`)**:
  - Checks if the shared folder exists; if not, creates it and returns.
  - Uses `os.walk` to traverse the `shared_folder_path`.
  - For each directory (excluding the root `.`), calculates its relative path and stores its modification time (`os.path.getmtime`) in `file_timestamps`.
  - For each file, calculates its relative path and stores its modification time in `file_timestamps`.
- **Recording Changes (`record_file_change`, `record_deletion`)**:
  - `record_file_change`: Updates the timestamp for the given `rel_path` in `file_timestamps`. If the path was previously in `deleted_items`, it removes it.
  - `record_deletion`: If the `rel_path` exists in `file_timestamps`, removes it. Adds the `rel_path` to the `deleted_items` set.
- **Getting Updates (`get_changes_since`)**:
  - Takes a `last_sync` timestamp as input.
  - Iterates through `file_timestamps`. If a path's timestamp is greater than `last_sync`, it adds the path and timestamp to an `updates` dictionary.
  - Returns a dictionary containing `{"updated": updates, "deleted": list(self.deleted_items)}`.

    - **Note:** The current implementation includes *all* items ever deleted in the `deleted` list, not just those deleted since `last_sync`. A more sophisticated implementation would track deletion timestamps.

3.2. Client Implementation (`client.py`)

3.2.1. `FileClient` Class

- **Initialization (`__init__`)**:
    - Stores server host/port, absolute local folder path, and poll interval.
    - Ensures the local folder exists (`os.makedirs(exist_ok=True)`).
    - Initializes `socket` to `None`, `last_sync_time` to 0.
    - Creates `FileSystemChangeHandler` instance.
    - Creates `watchdog.observers.Observer` instance.
    - Creates a `threading.Lock` (`sync_lock`).
    - Sets `running` flag to `False`.
    - Creates the `poll_thread` (target=`_poll_server_thread`) but doesn't start it yet.
- **Starting (`start`)**:
    - Calls `_connect_to_server`.
    - Sets `last_sync_time` to the current time.
    - Calls `_full_sync_with_server` to get the initial state.
    - Schedules the `event_handler` with the `observer` to monitor the `local_folder_path` recursively.
    - Starts the `observer` thread (`observer.start()`).
    - Logs the start of monitoring.
    - Sets `running` flag to `True`.
    - Starts the `poll_thread`.
    - Logs client start. Includes error handling to call `stop()` if startup fails.
- **Stopping (`stop`)**:

  - Logs the stop sequence.
  - Sets `running` flag to `False`.
  - Stops the `observer` thread if alive (`observer.stop()`, `observer.join()`).
  - Waits for the `poll_thread` to finish if alive (`poll_thread.join(timeout=1)`).
  - Calls `_disconnect_from_server`.
  - Logs client stop.
- **Connection Management (`_connect_to_server`, `_disconnect_from_server`)**:
  - `_connect_to_server`: Creates a new TCP socket, connects to the server address, sends a `CONNECT` message, waits for an `ACK`, and logs success or raises an error. Sets `self.socket`.
  - `_disconnect_from_server`: If `self.socket` exists, sends a `DISCONNECT` message, waits briefly for an ACK (with timeout), closes the socket, and sets `self.socket` to `None`. Includes error handling.
- **Polling Thread (`_poll_server_thread`)**:
  - Runs `while self.running`.
  - Inside the loop:
    - **Check for Incoming Data:** Uses `_socket_has_data()` (non-blocking check using `select.select`) to see if the server has sent anything proactively (like a broadcast). If data exists, acquires `sync_lock` and calls `_process_incoming_message`.
    - **Sleep:** Pauses for `self.poll_interval` seconds.
    - **Periodic Sync:** If still running, acquires `sync_lock`.
      - If connected (`self.socket` is not None), calls `_sync_with_server` (which sends `REQUEST_UPDATES`).
      - If not connected, attempts `_connect_to_server()`. If successful, calls `_full_sync_with_server()` to resynchronize completely. Logs reconnection failures.
    - **Error Handling:** Catches exceptions within the loop. If a `ConnectionError` or `OSError` occurs, sets `self.socket = None` to trigger reconnection logic in the next iteration.

- **Processing Server Messages (`_process_incoming_message`)**:
    - Called by the polling thread when data is available on the socket.
    - Receives one message using `protocol.receive_message`.
    - Extracts `action` and `payload`.
    - Handles various actions received *from the server*:
        - `CREATE_FILE`/`UPDATE_FILE`: Extracts path and data. Tells `event_handler` to ignore the upcoming local change (`ignore_next_event_for`). Creates parent directories. Uses `protocol.decode_file_data` to write the file locally. Logs the update.
        - `DELETE_FILE`: Extracts path. Calls `_apply_remote_deletion`.
        - `CREATE_FOLDER`: Extracts path. Calls `_apply_remote_folder_creation`.
        - `DELETE_FOLDER`: Extracts path. Calls `_apply_remote_deletion`.
        - `SEND_UPDATES`: (Response to periodic poll) Extracts update payload. Calls `_process_updates`.
        - `SEND_ALL_FILES`: (Response to initial sync request) Extracts payload. Calls `_process_all_files`. Updates `self.last_sync_time` based on the `sync_start_time` received from the server (or current time as fallback).
        - `ACK`: Logs the received acknowledgment.
        - `ERROR`: Logs the server error message.
        - Unknown actions are logged as warnings.
    - Includes error handling; sets `self.socket = None` on connection errors.
- **Periodic Sync Logic (`_sync_with_server`, `_request_updates`)**:
    - `_sync_with_server`: Calls `_request_updates` and then updates `self.last_sync_time` to the current time. (Note: `last_sync_time` ideally should be updated *after* successfully processing the response, which happens asynchronously via `_process_incoming_message` and `_process_updates`/`_process_all_files`).
    - `_request_updates`: Creates `REQUEST_UPDATES` message with the current `last_sync_time` and sends it to the server. *Does not wait for the response*. Includes error handling, setting `self.socket = None` on connection errors.

- **Processing Updates (`_process_updates`)**:
    - Called when a `SEND_UPDATES` message is received.
    - Processes `deleted` items first by calling `_apply_remote_deletion` for each path.
    - Processes `updated` items:
        - If the path indicates a directory (simple check for trailing slash - might be fragile), calls `_apply_remote_folder_creation`.
        - If it's a file, calls `_request_file_content`. **(Note: `_request_file_content` is currently a placeholder/FIXME, as the protocol doesn't define a way for the client to explicitly request specific file content after receiving an update notification. The current implementation relies on the server sending the full file content in the broadcast/initial sync).**
- **Applying Remote Changes (`_apply_remote_deletion`, `_remove_directory_recursive`, `_apply_remote_folder_creation`)**:
    - These methods modify the local filesystem based on server instructions.
    - Crucially, before performing any `os` operation (remove, rmdir, makedirs), they call `self.event_handler.ignore_next_event_for(abs_path)` to prevent `watchdog` from detecting this change and sending it back to the server as a local modification.
    - `_apply_remote_deletion`: Determines if the path is a file or directory. If it's a directory, calls `_remove_directory_recursive`. If it's a file, calls `os.remove`.
    - `_remove_directory_recursive`: Uses `os.walk(topdown=False)` to delete contents first, then the directory, ensuring `ignore_next_event_for` is called for every item being removed.
    - `_apply_remote_folder_creation`: Creates the directory using `os.makedirs(exist_ok=True)`.
- **Handling Local Changes (`handle_local_creation`, `handle_local_deletion`, `handle_local_modification`)**:
    - These methods are called by `FileSystemChangeHandler` when `watchdog` detects a change.
    - They verify the change occurred within the managed `local_folder_path`.
    - Calculate the relative path (`rel_path`).

    - Acquire the `sync_lock`.
    - Check if connected (`self.socket`).
    - Create the appropriate protocol message (`CREATE_FILE`, `DELETE_FOLDER`, etc.) using helper functions in `protocol.py`. For file creation/modification, `protocol.encode_file_data` is used.
    - Send the message to the server using `protocol.send_message`.
    - Log the action.
    - *Do not wait for an ACK* (ACKs are handled asynchronously by `_process_incoming_message`).
    - Include error handling, setting `self.socket = None` on connection errors.
    - Release the `sync_lock`.
- **Full Sync (`_full_sync_with_server`, `_process_all_files`)**:
    - `_full_sync_with_server`: Sends `REQUEST_ALL_FILES` message. Waits for and receives the `SEND_ALL_FILES` response. Calls `_process_all_files` with the payload. Updates `last_sync_time` based on the server's `sync_start_time`. Handles potential errors and unexpected responses.
    - `_process_all_files`: Receives the dictionary containing `folders` and `files` lists. Iterates through `folders`, calling `self.event_handler.ignore_next_event_for` and `os.makedirs` for each. Iterates through `files`, calling `ignore_next_event_for`, ensuring parent directories exist, and using `protocol.decode_file_data` to write each file locally.

3.2.2. `FileSystemChangeHandler` Class (Watchdog Handler)

- **Initialization (`__init__`)**:
    - Stores the `FileClient` instance.
    - Initializes `_ignore_next_events` as an empty set to store absolute paths of events that should be ignored.
- **Ignoring Events (`ignore_next_event_for`)**: Adds a given absolute path to the `_ignore_next_events` set.
- **Event Callbacks (`on_created`, `on_deleted`, `on_modified`, `on_moved`)**:

    - These methods are called by the `watchdog` observer thread when filesystem events occur.
    - **Path Normalization:** Ensures the event path (`event.src_path`, `event.dest_path`) is a string using `_ensure_str`.
    - **Ignore Check:** Checks if the `src_path` is in `_ignore_next_events`. If yes, removes it from the set and returns immediately (this prevents processing self-induced changes).
    - **Temporary File Check:** Calls `_is_temp_file` to check if the path corresponds to common temporary file patterns. If yes, returns immediately.
    - **Delegation:** If the event should be processed, calls the corresponding `self.client.handle_local_...` method, passing the absolute path(s) and whether it's a directory (`event.is_directory`).
    - **Move Handling (`on_moved`):** Currently implemented simply as a delete of the source path followed by a create of the destination path.
- **Utility Methods (`_is_temp_file`, `_ensure_str`)**:
    - `_is_temp_file`: Checks if a filename matches common temporary/system file patterns (e.g., `~$`, `.swp`, `.DS_Store`).
    - `_ensure_str`: Converts potential bytes/bytearray paths from `watchdog` into UTF-8 strings.

3.3. Protocol Implementation (`protocol.py`)

- **Constants:** Defines action strings (`ACTION_CREATE_FILE`, etc.), status codes, `MAX_MESSAGE_SIZE`, and `HEADER_SIZE`.
- **Message Creation (`create_message`)**:
    - Takes an `action` string and an optional `payload` dictionary.
    - Constructs the main message dictionary `{"action": action, "payload": payload}`.
    - Serializes this dictionary to a JSON string and encodes it to UTF-8 bytes (`json_data`).
    - Calculates the length of `json_data`.

    - Creates the 8-byte header by converting the length to bytes (`to_bytes(HEADER_SIZE, byteorder='big')`).
    - Returns the concatenated `header + json_data`.
- **Message Parsing (`parse_message`)**:
    - Takes the raw message bytes (including the header).
    - Slices the bytes to get only the JSON payload (`message_bytes[HEADER_SIZE:]`).
    - Decodes the JSON bytes back into a UTF-8 string and parses it into a Python dictionary using `json.loads`.
    - Returns the parsed dictionary.
- **File Data Handling (`encode_file_data`, `decode_file_data`)**:
    - `encode_file_data`: Checks if the file exists and is non-empty. Reads the file in binary mode (`'rb'`). Encodes the bytes using `base64.b64encode` and decodes the resulting Base64 bytes into a UTF-8 string for JSON compatibility. Returns `None` for empty or non-existent files.
    - `decode_file_data`: Ensures the target directory exists (`os.makedirs(..., exist_ok=True)`). If `encoded_data` is `None`, creates an empty file. Otherwise, decodes the Base64 string back into bytes (`base64.b64decode`) and writes these bytes to the target file in binary mode (`'wb'`).
- **Action-Specific Message Builders (`create_file_message`, `create_delete_folder_message`, etc.)**:
    - These are convenience functions that construct the correct payload dictionary for a specific action and then call `create_message`.
    - They handle gathering necessary information like relative paths, absolute paths (for reading content), timestamps (`os.path.getmtime` or `time.time()`), and calling `encode_file_data` where needed.
- **Socket Communication (`receive_message_header`, `receive_message`, `send_message`)**:
    - `receive_message_header`: Reads exactly `HEADER_SIZE` bytes from the socket, handling potential partial reads in a loop. Returns the message size integer. Raises `ConnectionError` if the socket closes prematurely.

  - `receive_message`: First calls `receive_message_header` to get the expected payload size. Checks if the size exceeds `MAX_MESSAGE_SIZE`. Reads the specified number of bytes for the payload from the socket, handling partial reads in a loop (`recv(min(4096, remaining_bytes))`). Reconstructs the full message (header + payload bytes) and calls `parse_message`. Raises `ConnectionError` or `ValueError`.
  - `send_message`: Takes the complete message bytes (header + payload). Sends the data over the socket using `sock.send`, handling partial writes in a loop until all bytes are sent. Raises `ConnectionError` if the socket connection breaks during sending (`sent == 0`).

3.4. Entry Points (`main_server.py`, `main_client.py`)

- **Argument Parsing (`parse_arguments`)**: Both use `argparse` to define and parse command-line arguments:
  - Server: `--host`, `--port`, `--folder` (for server's shared folder).
  - Client: `--host`, `--port` (for server address), `--folder` (for client's local folder), `--poll` (polling interval). Default values are provided.
- **Signal Handling (`handle_signal`)**: Both implement a simple signal handler for `SIGINT` (Ctrl+C) and `SIGTERM` to print a shutdown message and exit gracefully (`sys.exit(0)`). This handler is registered using `signal.signal`.
- **Main Function (`main`)**:
  - Parses arguments.
  - Prints startup information (addresses, folders).
  - Initializes the respective core class (`FileServer` or `FileClient`) with the parsed arguments.
  - Registers signal handlers.
  - **Server:** Calls `server.start()` within a `try...finally` block. The `finally` ensures `server.stop()` is called on exit (normal or exception). Catches `KeyboardInterrupt` for clean shutdown message.

    - **Client:** Calls `client.start()` within a `try...finally` block. The `finally` ensures `client.stop()` is called. Enters an infinite `while True: time.sleep(1)` loop after starting the client. This keeps the main thread alive while the client's background threads (observer, poller) do the work. Catches `KeyboardInterrupt` to break this loop and proceed to the `finally` block for cleanup.

- **Execution Guard (`if __name__ == "__main__":`)**: Ensures the `main()` function is called only when the script is executed directly.

4. Synchronization Strategy Analysis

The system employs a hybrid synchronization approach combining real-time updates with periodic polling:

1. **Initial Sync:** When a client connects (`ACTION_REQUEST_ALL_FILES`), the server sends the entire current state (`ACTION_SEND_ALL_FILES`). This establishes a baseline for the client. The server includes a `sync_start_time` timestamp in the response, indicating when it began scanning the directory. The client uses this timestamp as its initial `last_sync_time`.
2. **Local Changes (Client -> Server):** The client's `watchdog` observer detects local filesystem changes. The `FileSystemChangeHandler` filters these events and calls the appropriate `FileClient.handle_local_...` method. This method sends a message (e.g., `ACTION_CREATE_FILE`, `ACTION_DELETE_FOLDER`) to the server detailing the specific change.
3. **Real-time Broadcast (Server -> Other Clients):** When the server's `ClientHandler` processes a change request from one client and successfully modifies its shared folder, it immediately calls the appropriate `FileServer.broadcast_...` method. This sends a notification message (e.g., `ACTION_UPDATE_FILE`, `ACTION_DELETE_FOLDER`) about that *specific* change to all *other* connected clients.
4. **Applying Broadcasts (Client):** Clients continuously listen for messages from the server in their `_poll_server_thread` (specifically via `_process_incoming_message`). When a broadcast message arrives, the client applies the change directly to its local filesystem (`_apply_remote_deletion`, `protocol.decode_file_data`, etc.), making sure to ignore the resulting `watchdog` event.

5. **Periodic Polling (Client -> Server):** To catch any updates missed during potential disconnections or broadcast failures, the client's `_poll_server_thread` periodically calls `_sync_with_server`. This sends an `ACTION_REQUEST_UPDATES` message to the server, including the client's `last_sync_time`.
6. **Polling Response (Server -> Client):** The server's `_handle_request_updates` uses the `FileChangeTracker` to find all items modified (`file_timestamps`) or deleted (`deleted_items`) since the client's `last_sync_time`. It sends this list back in an `ACTION_SEND_UPDATES` message.
7. **Applying Polled Updates (Client):** When the client receives `ACTION_SEND_UPDATES` (via `_process_incoming_message`), it calls `_process_updates`. This iterates through the `deleted` list (calling `_apply_remote_deletion`) and the `updated` list. For updated items, it currently relies on having received the content from a broadcast or needing a mechanism (like the FIXME `_request_file_content`) to fetch the data. After processing, the client updates its `last_sync_time`.

Efficiency: This hybrid model aims for efficiency. Real-time broadcasts provide low latency for changes. Polling acts as a fallback mechanism. Transmitting only changes (deltas) after the initial sync minimizes bandwidth usage compared to repeatedly sending the full state. However, the current polling response for *updates* might require a follow-up request from the client to get actual file content if it wasn't received via broadcast. The broadcast mechanism itself sends the full file content on every update.

Limitations:

- **No Conflict Resolution:** If two clients modify the same file concurrently, the server simply processes the requests as they arrive. The last client's change to reach the server will overwrite previous ones, and this last change will be broadcast.
- **Deletion Tracking:** The server's `FileChangeTracker` currently returns *all* deleted items in polling responses, not just those deleted since the last sync.
- **Update Content Fetching:** The client logic for handling `updated` items in the polling response (`_process_updates`) assumes it either already has the content from a broadcast or needs a (currently unimplemented) way to request the specific file content.

5. Potential Improvements and Future Work

- **Conflict Resolution:** Implement mechanisms to detect concurrent edits (e.g., using version vectors or comparing timestamps more carefully) and handle conflicts (e.g., creating duplicate files, prompting the user, or implementing a merge strategy).
- **Efficient Updates (Delta Sync):** For text files, implement delta synchronization (sending only the changes/diffs) instead of the entire file content on each update to save bandwidth. Libraries like `diff-match-patch` could be explored.
- **Binary File Deltas:** Explore binary diffing algorithms for more efficient updates of large binary files.
- **Improved Deletion Tracking:** Modify `FileChangeTracker` to store deletion timestamps, allowing polling responses to include only relevant deletions.
- **Robust Update Fetching:** Implement `ACTION_REQUEST_FILE_CONTENT` and corresponding server handling so clients can explicitly request file data identified in `SEND_UPDATES` responses if needed.
- **Error Handling and Resilience:** Add more robust error handling for network issues, filesystem errors, and unexpected message formats. Implement retry mechanisms for failed operations.
- **Security:** Integrate TLS/SSL for encrypted communication. Add user authentication and authorization mechanisms.
- **Scalability:** For a larger number of clients, consider asynchronous I/O (e.g., `asyncio`) instead of threads to handle connections more efficiently. Optimize locking mechanisms.
- **Configuration:** Move settings like port, host, and folder paths to configuration files instead of only command-line arguments.
- **Testing:** Develop comprehensive unit and integration tests to ensure correctness and robustness.
- **GUI:** Create a graphical user interface for easier user interaction.

6. Conclusion

This Python-based Dropbox-like simulator successfully demonstrates core file synchronization concepts using fundamental socket programming. The client-server architecture, combined with the defined JSON-based protocol and a hybrid synchronization strategy (broadcasts and polling), allows multiple clients to maintain a reasonably consistent view of a shared folder.

The implementation details, including multi-threaded client handling on the server, `watchdog`-based local change detection on the client, and the careful handling of message framing and data serialization, provide a solid foundation. While lacking advanced features, security, and robust conflict resolution found in production systems, this project serves as a valuable educational tool for understanding the principles behind distributed file synchronization over a network. The identified limitations and potential improvements offer clear directions for future development.