**Project Documentation**

I have neither given nor received unauthorized assistance on this work. I will not post the project description and the solution online.

Sign: Chandra Vamsi Krishna Alla                                       Date: 06/29/2024

**Name:** Chandra Vamsi Krishna Alla.

**ID:** 1002170054

**Introduction**

This project involves developing a multi-threaded file server and a computational server that communicate using Remote Procedure Call (RPC). The documentation outlines the technologies used, implementation details, and operational explanations.

**Technologies Used**

The project utilizes the Python programming language, with libraries such as xmlrpc and rpyc for RPC communication and filecmp for file comparison.

**Implementation:**

**Part 1: Multi-Threaded File Server**

In Part 1, we created a multi-threaded file server supporting the following operations: Upload, Download, Delete, and Rename.

**Upload Operation:**

This operation transfers a file from the client to the server.

Both the client and server have separate directories for file storage.

After selecting the upload operation, the user provides the filename to be uploaded to the server from the client.

**Download Operation**

This operation allows users to download a file from the server to the client.

After selecting the download operation, the user specifies the filename to be downloaded to the client from the server.

**Delete Operation**

The delete operation removes a file from the server.

After selecting the delete operation, the user provides the filename to be deleted from the server.

**Rename Operation**

The rename operation changes a file's name on the server.

After selecting the rename operation, the user specifies the current filename to be renamed on the server and the new filename.

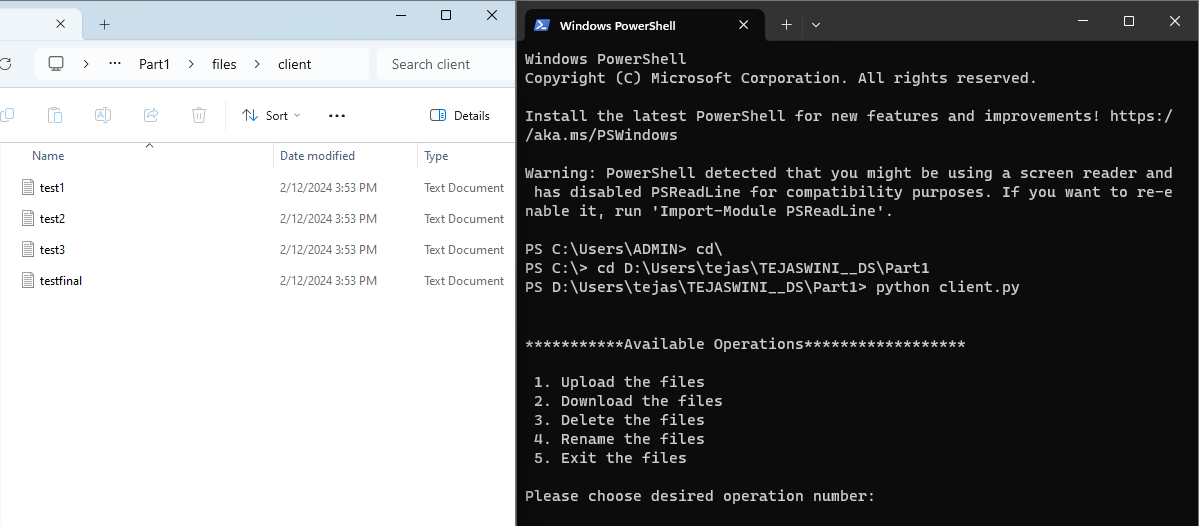
**Code Overview:**

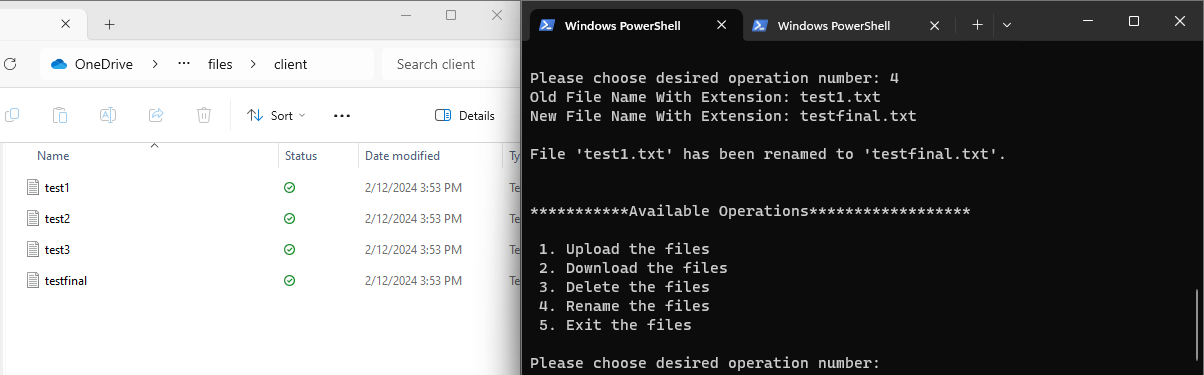
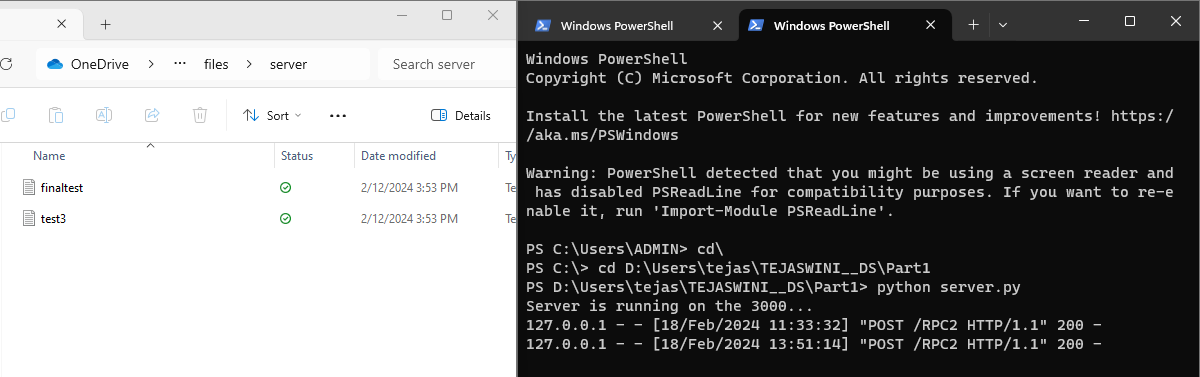
**Client:**

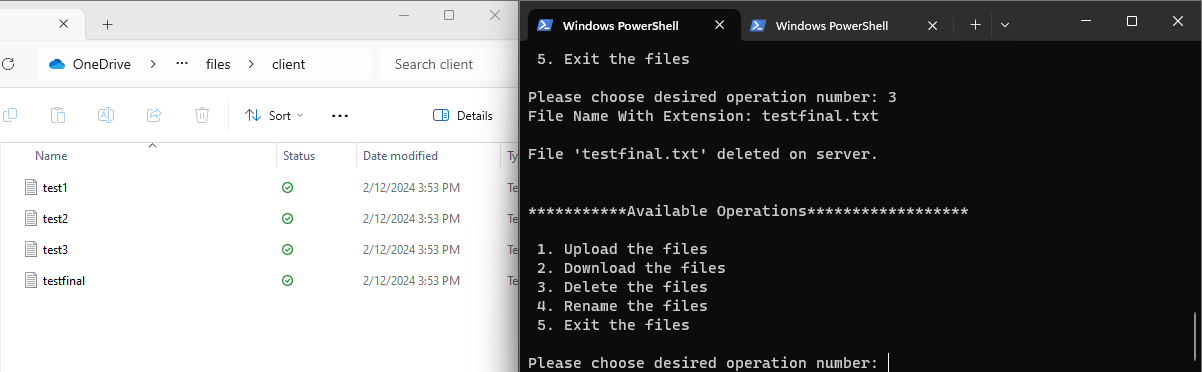
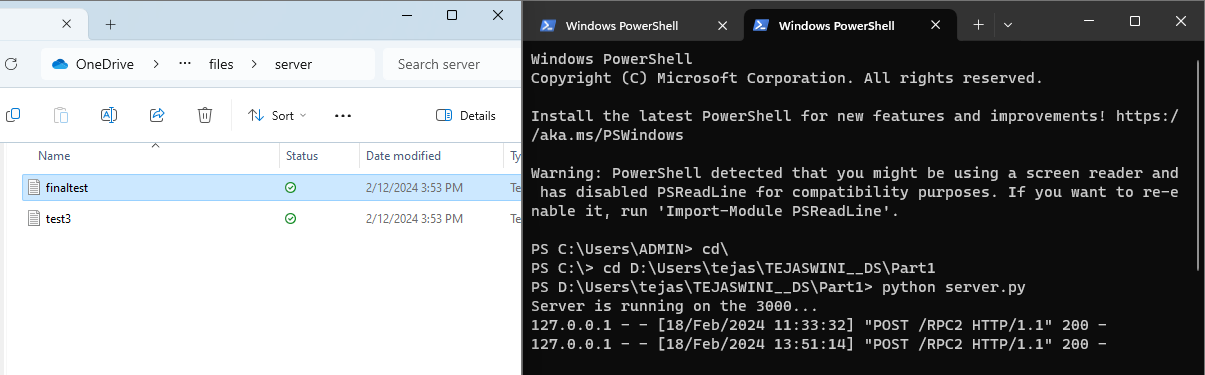
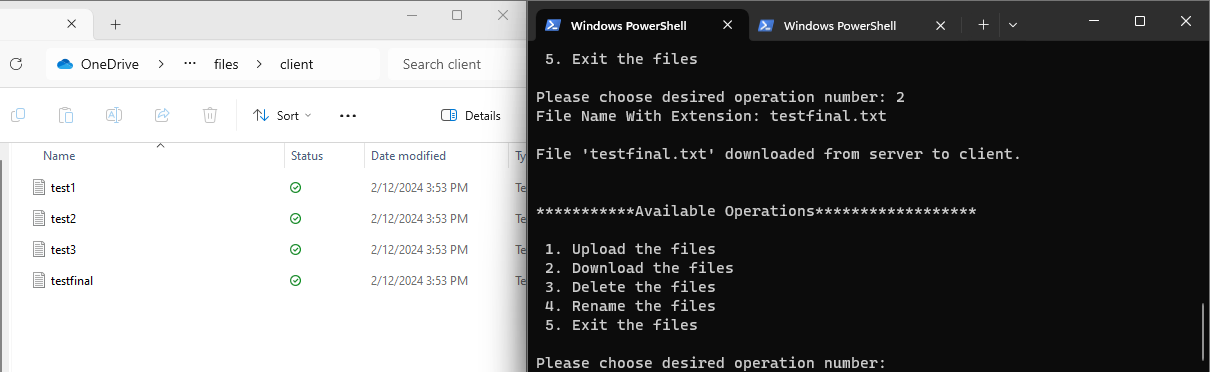
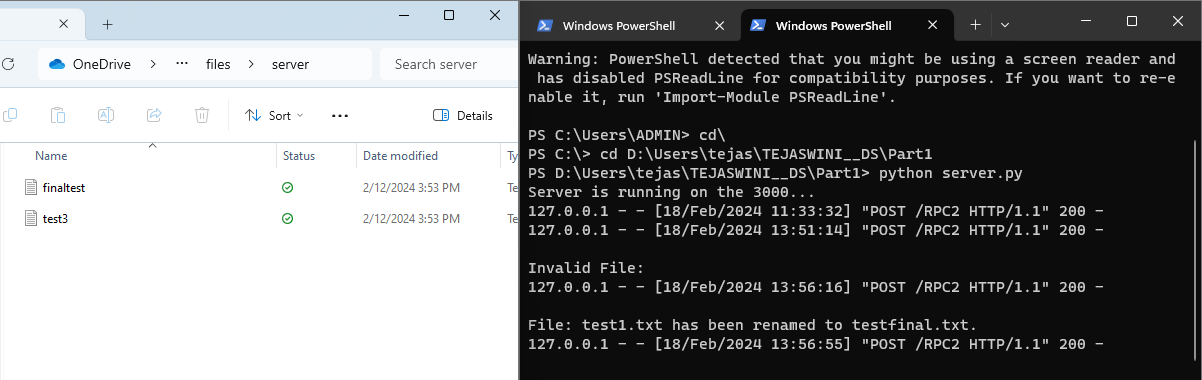
* Import the essential libraries and modules, such as xmlrpc.client, os, and threading, to connect to the server, manage file operations, and support multi-threading.
* Define constants such as the server port number and the location where client files are saved.
* The ServerProxy class can be used to create an XML-RPC client proxy that communicates with the server. It connects to the server using the provided port.
* Define functions for different file operations:
* UploadFileToServerPart1(fileName): Transfers a file from the client to the server.
* downloadFileFromServerPart1(fileName) downloads a file from the server to the client.
* deleteFileFromServerPart1(name\_f) deletes a file on the server.
* The function renameFileInServerPart1(name\_f\_old, name\_f\_new) renames a server file.
* Create a function called startClientExecution() to handle user input and perform operations based on the selected choice.
* Create a thread masterThread for client execution, which allows the client to do numerous actions concurrently.

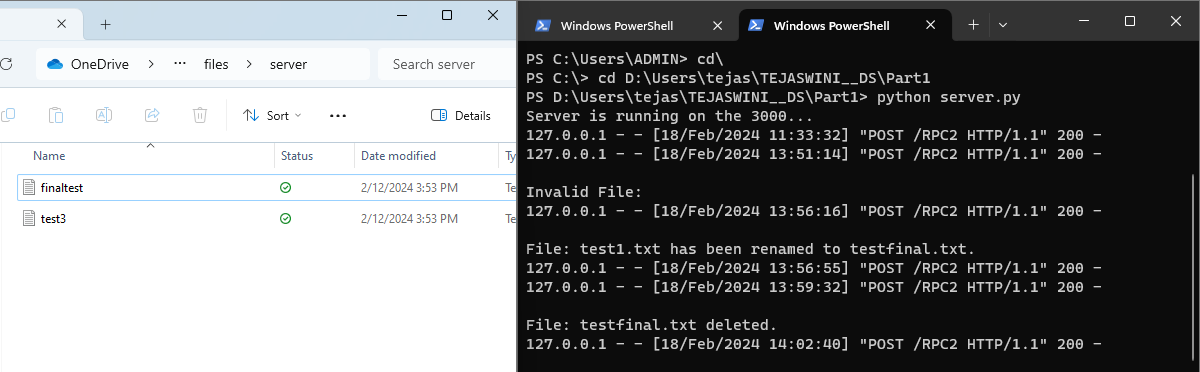
**Server:**

* Import the essential libraries and modules, such as SimpleXMLRPCServer to build an XML-RPC server, ThreadingMixIn to enable multi-threading, xmlrpc.client, and os.
* Define constants such as the server's port number and the place where server files are saved.
* Create a custom class called CustomXMLRPCServer, which inherits from CreateThread and SimpleXMLRPCServer. This custom class supports multi-threading on the server.
* Implement methods that handle file operations on the server:
* The function uploadFileToServerPart1(name\_file, directory\_file) uploads a file to the server.
* downloadFileFromServerPart1(name\_file): Obtains a file from the server.
* deleteFileFromServerPart1(name\_file) deletes a file from the server.
* renameFileInServerPart1(name\_file\_old, name\_file\_new): Renames a server-side file.
* Make an instance of the custom server class, register any required functions, and specify the server's IP and port number.
* Start the server using server.serve\_forever(), which keeps it running and ready to answer client queries.









**Part 1**

**Part 2: Automatic File Syncing**

Part 2 comprises automatic file synchronization between the client and server. When a client creates, updates, or deletes a file, a helper thread duplicates the changes on the server.

Every 4 seconds, the helper thread checks for file changes in the client directory and updates the server directory if any are identified.

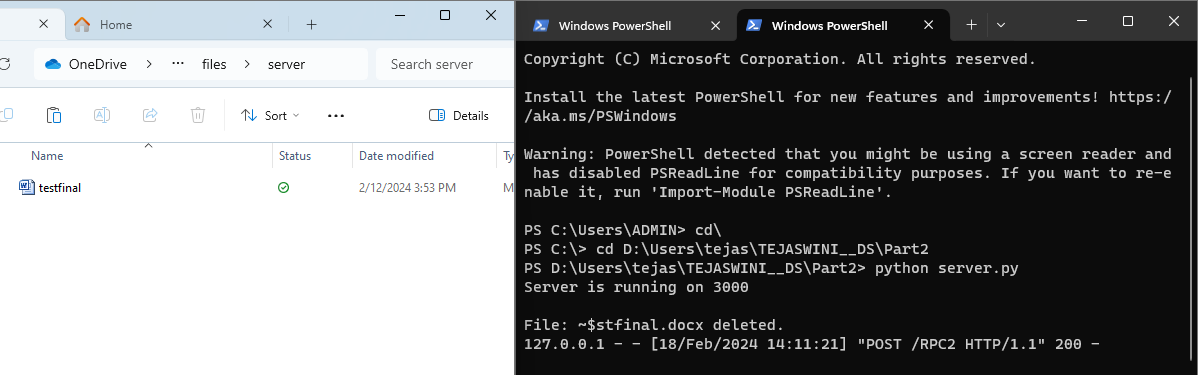
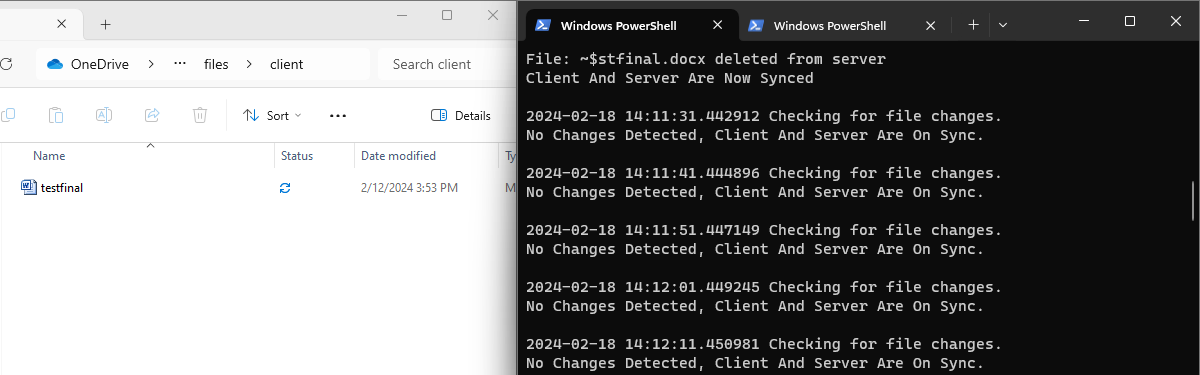
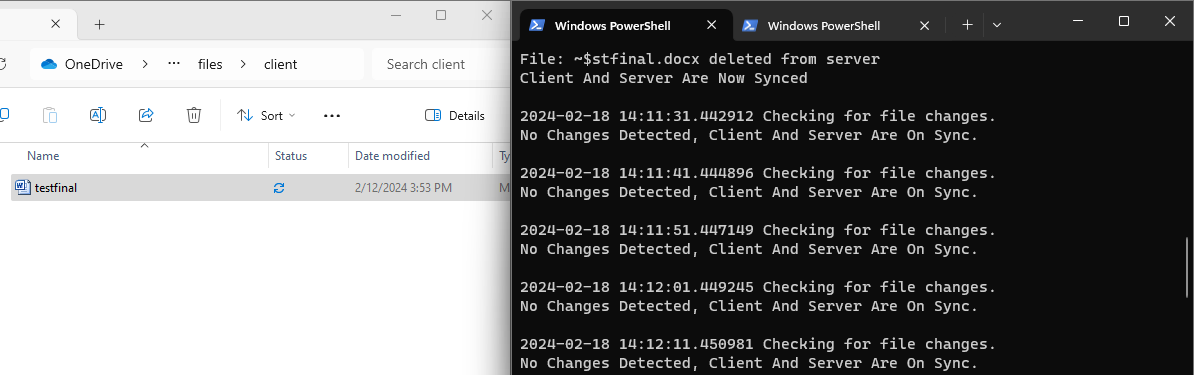
**Code Overview:**

**Client Code:**

* Imports the Python libraries required for XML-RPC communication, timestamp management, multithreading, file comparison, and file operations.
* Defines constants, including the server's port number, and generates an XML-RPC client proxy for connecting to the server.
* Specifies the paths to the client and server's file directories where files will be saved.
* Implements a function (uploadFileToServerPart2) that uploads files to the server and handles file upload failures.
* Implements a function (deleteFileFromServerPart2) that deletes files from the server while handling errors for unsuccessful deletions.
* Creates a function (automaticDirectorySync) that keeps directories synchronized between the client and server. It distinguishes between server-only files for deletion, client-only files for upload, and modified client files for upload. Gives comments on synchronization.
* Uses a helper thread to perform the automaticDirectorySync method concurrently, constantly checking for file changes between the client and server directories.

**Server Code:**

* Imports Python libraries, such as SimpleXMLRPCServer for XML-RPC, ThreadingMixIn for multi-threading, and os for file operations.
* Defines server-specific constants like port number and directory path.
* Creates a custom class (CustomXMLRPCServer) that inherits from CreateThread and SimpleXMLRPCServer to enable multi-threading on the XML-RPC server.
* Added a function (uploadFileToServerPart2) to manage file uploads to the server.
* It creates the entire path to the server file, stores the submitted binary data, and handles faulty file uploads.
* Added deleteFileFromServerPart2 function to handle server-side file deletions.
* It removes the provided file from the server directory and returns a success or error message depending on the outcome.
* Initializes the server by creating a custom server class instance, registering introspection functions, registering the uploadFileToServerPart2 and deleteFileFromServerPart2 methods, and launching the server.



**Part 2**

**Part 3: Computation Server**

In Part 3, we built a computation server that supports two operations: adding two numbers and sorting an array via both synchronous and asynchronous RPCs.

**Synchronous RPC**

In synchronous RPC, the client requests an operation from the server and expects an immediate answer.

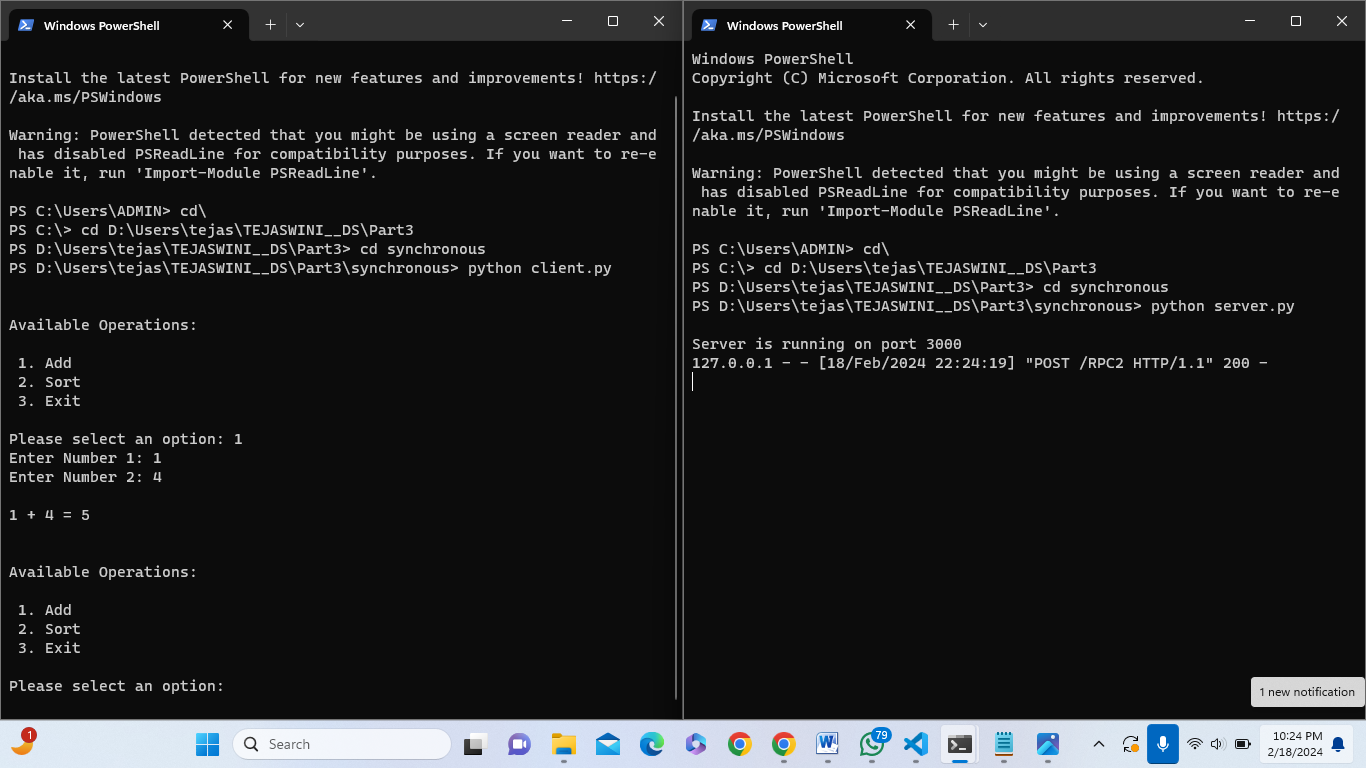
**Code Overview:**

**Client:**

* Imports xmlrpc.client, os, and threading.
* Defines the server port as 3000.
* Creates an XML-RPC client proxy (rpc\_server) for communicating with the server.
* Implements the additionSync function for addition operations and the sortSync function for sorting operations on the server.
* Handles user input to choose and execute operations (Add or Sort).
* Client activities are handled concurrently on a different thread (main\_thread).

**Server:**

* The following libraries are imported: SimpleXMLRPCServer and ThreadingMixIn.
* Defines the server port as 3000.
* Creates a custom XML-RPC server class (CustomXMLRPCServer) that inherits from CreateThread and SimpleXMLRPCServer.
* Implements the additionSync and sortingSync functions for addition and sorting operations, respectively.
* Creates a custom server class, registers functions, and defines server information (localhost and port).
* Registers standard introspection functions, gives the additionSync and sortingSync methods appropriate names, and begins processing client requests indefinitely.



**Part 3 Synchronous**

**Asynchronous RPC**

In asynchronous RPC, the client requests an action and waits for a response from the server. It can carry out other duties while the server handles the job. When completed, the server transmits the response to the client.

**Code Overview:**

**Client:**

* Import the required libraries, which include asyncio for asynchronous tasks, rpyc for RPC communication, and os for system-related operations.
* Define the server's port number as server\_port, which is 3000.
* Connect to the server via rpyc.connect() to initiate contact with the server running on localhost at the specified port.
* Create an asynchronous function that simulates background processes. It executes a loop for 10 iterations, prints a message, and then sleeps for two seconds between iterations.
* Define the execute\_client\_operations() function, which manages user interactions. It shows a list of available operations (Add, Sort, and Exit) and waits for user input.
* If the user clicks "Add," it prompts for two numbers, makes an asynchronous call to server\_connection.root.addFunction, transmits the numbers to the server for addition, then asynchronously checks for the result status, which can take up to 5 seconds.
* If the user chooses "Sort," the program prompts for a comma-separated list of numbers, makes an asynchronous call to server\_connection.root.sortFunction, transmits the list to the server for sorting, then checks for the result status asynchronously, waiting up to 1 second.
* If the user clicks "Exit," the software terminates.
* The main\_function() function executes execute\_client\_operations() and simulate\_background\_tasks() simultaneously using asyncio.gather() in an infinite loop.

**Server:**

* Import the essential libraries, such as rpyc for RPC communication, ThreadedServer from rpyc.utils.server to create a threaded server, and time for time-related operations.
* Define the server's port number as 3000.
* Use RPyC's Service class to create a custom XML-RPC server called CustomXMLRPCServer.
* Implement the on\_connect method, which is invoked when a client connects to the server and prints a message to confirm the connection.
* Implement the on\_disconnect method, which is invoked when a client disconnects from the server and produces a message to indicate the disconnection.
* For remote procedure calls, expose two methods to clients: exposed\_addFunction and exposed\_sortFunction. These methods conduct addition and sorting operations, respectively, and include print statements to monitor their progress.
* Define asynchronous functions additionAsync and sortAsync to perform addition and sorting operations, respectively.
* ThreadedServer creates a threaded server instance named server. It uses the CustomXMLRPCServer class to listen on the specified port.
* Start the server by calling server.start(), which listens for incoming client connections and handles their RPC queries.

**Learnings:**

* RPC Protocol Knowledge: I gained a thorough understanding of Remote Procedure Call (RPC) protocols, particularly how they permit communication between distributed systems.
* Project Architecture: I learned how to analyze and describe complicated project structures, such as component divisions and interactions.
* File Operations: Learned about file handling operations in a client-server architecture, including UPLOAD, DOWNLOAD, DELETE, and RENAME.
* Multithreading: I've mastered multithreading principles and techniques, especially for managing concurrent client requests and server-side file synchronization.
* Synchronization Strategies: I learned about file synchronization strategies like real-time updates and timestamp-based reasoning.
* RPC Libraries: Gained experience utilizing RPC libraries such as xmlrpc and rpyc to implement client-server communication.
* Error Handling: Learned how to handle errors and manage exceptions, assuring the reliability of both client and server components.
* Asynchronous Programming: I learned how to efficiently implement asynchronous RPC calls, which allow for asynchronous client-server interactions.

**Challenges Faced:**

We encountered difficulties during the implementation of RPC. Understanding and managing the relationship between the server and client. Understanding which ports are open and why a few are unable to open. Determine whether two directories are comparable or whether they have been modified. Identifying effective strategies for managing file uploads and downloads on the server.