**Teammate 1: Shruti Borge (1002122976)**

**Agreement of Understanding: I have neither given nor received unauthorized assistance with this work.**

**Sign: Shruti Borge**

**Date: 29 June 2024**

**CSE-5306-001 Distributed Systems Project 1**

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

**Introduction:**

The objective of this section was to implement a multi-threaded file server capable of handling four fundamental file operations: RENAME, DELETE, DOWNLOAD, and UPLOAD. This server provides a simple and efficient way to manage files on the server, allowing users to interact with it via a client application. Both the server and client components are written in Python using the 'xmlrpc' package for communication. This project serves as an educational exercise in networked file management, enhancing our understanding of Python programming and network communication techniques.

**Server Functions:**

**create\_folders (upload\_folder, download\_folder)**: This function creates the necessary upload and download directories on the server if they do not already exist, using the `os.makedirs` function to ensure the folders are created safely.

**upload\_file(file\_name, file\_data)**: Initially, this function had an issue where the uploaded file data was being printed to the command line instead of being written to a file. By opening the file in binary write mode ("wb"), this problem was resolved, enabling successful file uploads.

**download\_file(file\_name)**: Allows clients to download files from the server. The file is read in binary mode and sent to the client as a binary object, with error handling to manage any issues.

**rename\_file(old\_file\_name, new\_file\_name)**: Enables clients to rename files on the server using the `os.rename` function, providing feedback to the client on the success or failure of the operation.

**delete\_file(file\_name)**: Allows clients to delete files from the server's upload folder using the `os.remove` function, returning the result to the client.

**Client Functions:**

**Upload a File:** Users can upload files by specifying the full file path. The client reads the file in binary mode and sends it to the server using `xmlrpc.client.Binary`.

**Download a File:** Allows users to download a file from the server by specifying the file name. The client saves the received binary data as a file in the local `client\_download` folder.

**Rename a File:** Users can rename a file on the server by specifying the current and new file names, sending the rename request to the server.

**Delete a File:** Enables users to remove a file from the server by providing its name, with a warning to ensure the correct and most recent name is used.

**Exit:** Exits the client application.

**Note**: We have also implemented multithreading using socket communication with files `server\_using\_socket.py` and `client\_using\_socket.py`, which perform the same operations as the server and client pair using `xmlrpc`. These files are for understanding sockets.

**Part 2: Dropbox-Like Synchronized Storage Service**

In Part 2 of the project, we used Flask and Python to further enhance our file synchronization service. This service provides real-time file synchronization between a client and a server, improving data accessibility and sharing across devices. Our project consists of two core components: the client-side script (`client.py`) and the server-side script (`server.py`).

Significant changes have been made to the client script, which now has a multithreaded architecture. We added a dedicated helper thread called `check\_sync()` that monitors file changes in a local directory (`cli\_sync\_folder`). When changes are detected, the helper thread queries the server and sends updated files to be stored in the server's directory (`ser\_sync\_folder`). These improvements, enabled by the Python `threading` module, allow for concurrent task execution, significantly enhancing our service's efficiency and responsiveness.

This implementation deepened our understanding of the importance of multithreading in real-time applications. Initially, we faced synchronization issues related to server IP configuration and file deletion, but these challenges were successfully overcome through meticulous debugging.

**Part 3: Synchronous and Asynchronous RPCs**

We established two server-client pairs: one for synchronous RPCs and another for asynchronous RPCs. Using the `xmlrpc` library, a basic XML-RPC server and client were created for the synchronous server-client pair. The client waits for the server's `add` and `sort\_array` functions to return results after invoking them via remote procedure.

For the asynchronous server-client pair, we used the `aiohttp` library to implement an asynchronous HTTP server and client. The server handles POST requests to two endpoints: `/add` and `/sort\_array`.

The main learning point was that synchronous RPCs block the client until the server completes the operation, whereas asynchronous RPCs allow the client to continue with other tasks while waiting for a response. The code can be further improved for scalability and security.

**Contributions**:

Part 1: Shruti

Part 2: Shruti

Part 3: Shruti

Report: Shruti