

Project Report: Distribution File Synchronization System

Sanskar Bajimaya / Amir Karki

Fairleigh Dickinson University – Vancouver Campus

INFO 4278 | Operating System

Professor (Dr.) Anita Saravana Kumar

Date: 07 / 04 /2025

Abstract

This report serves to implementation of a Linux-based distribution of File in Synchronization (Dropbox-like) across multiple users under same group. The project implemented the use of TCP/IP protocol to transfer data and communicate in local networking. Each client under the same group will have access to a file “client_files” where they can work on same file avoiding conflict. The project also allows version control through server as each version of files will be saved on the “server files” on the server. The client cannot directly access each version of file, but the owner (server) can provide every version of a file from server files to the client manually. Additionally, the File Watcher and Server will monitor each modification in the client files including the time stamps as logs to have all information of each modification.

Introduction

Motivation

There is multiple commercial system where multiple clients sync files with a central server using TCP/IP. Examples include Dropbox which is a cloud file sync using delta encoding with real time sync, Rsync which is a Linux tool for effective file transfer using partial/delta sync, Owncloud which is self-hosted file-sharing and similar to server-client model and many more. This project tends to implement real-time sync via TCP/IP protocol, handle conflicts and security, using client-server model.

Objective

The project’s main objective can be classified into 2 stages and additional future work:

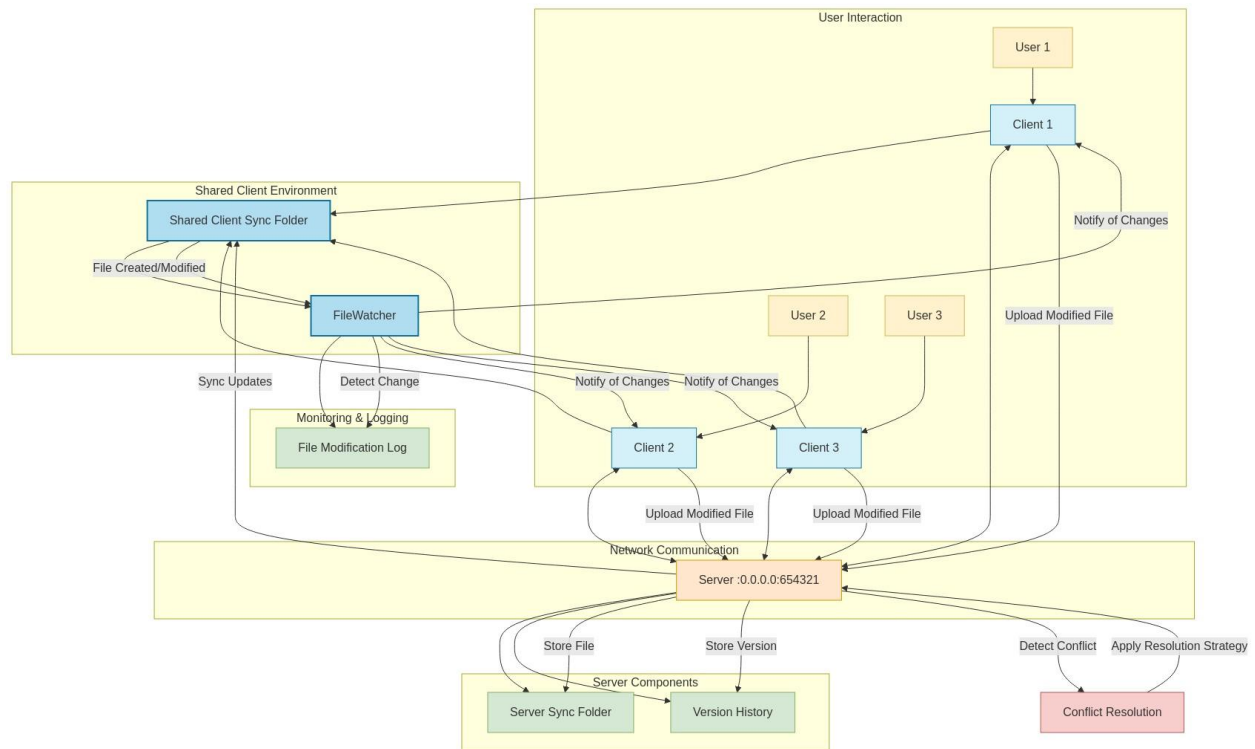
- Real-time sync via TCP/IP.
- Handle conflicts and Security.
- Optimize transfers (delta sync)

Scope

The scope of this project is to include Linux users as clients, Linux owner as a centralized server with conflict resolution and version control. The limitations of this project is absence of delta sync for better optimization and performance, absence of mobile support and accessibility issues.

System Design Architecture

Client-Server model



Access by Multiple Users

Through their client applications, many users establish connections. Every client uses the same shared sync folder.

Change Recognition

The shared folder is watched for any modifications by the lone FileWatcher. FileWatcher recognizes and records any user modifications to a file.

Synchronization

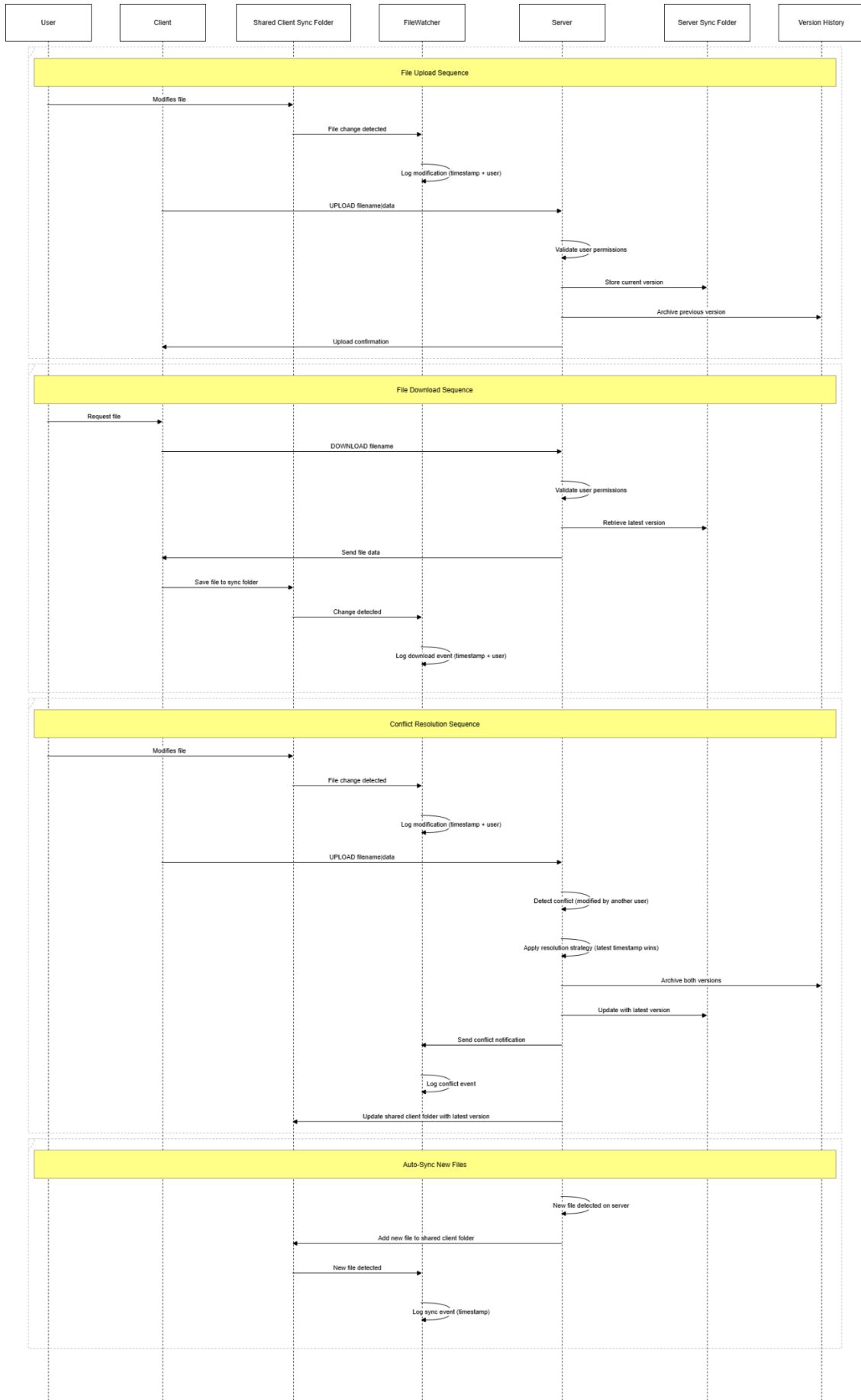
Clients use 0.0.0.0:654321 to communicate with the server. Changes are automatically synchronized in both directions by the server. The server keeps track of every file version in its version history.

Handling Conflict

The server recognizes conflicts when changes are made concurrently. A resolution technique (such as timestamp-based) is used.

Distribution File Synchronization System

4



Sequence of File Uploads

1. The user edits a file. A user edits a file located in the shared client sync folder.
2. FileWatcher identifies the modification. The FileWatcher component merely logs this event with the timestamp and user details when it detects that a file has been altered. FileWatcher simply maintains a record without informing anyone.
3. The client uploads the file; the client program uses a **UPLOAD** command with the filename and file data; it recognizes the change and transmits the updated file to the server.
4. The server handles the upload, verifies that the user is authorized to edit the file, saves the most recent version in its sync folder and archives the earlier version in its version history.

Sequence of File Downloads

1. A file is requested by the user: A user requests a particular file from their client application. The client asks the server for the file.
2. The client gives the server a **DOWNLOAD** command along with the filename. The file is retrieved by the server.
3. The server verifies that the user is authorized to access this file. The server receives the most recent file version. The client receives the file data from the server.
4. The client locally saves the file. The received file is saved by the client in the shared sync folder.
FileWatcher detects this change. FileWatcher records the download event together with the user's information and timestamp.

Sequence of Conflict Resolution

1. When a user makes changes to a file, user makes changes to a file that has previously been altered by another user. FileWatcher records this event of modification.
2. When the client uploads the updated file, the client uses the **UPLOAD** command to transfer the file to the server.
3. The server resolves the dispute. The server notices that several users have altered this file. The "latest timestamp wins" rule is automatically applied by the server, which saves both versions in its version history. The server adds the most recent version to its own sync folder. FileWatcher receives a conflict notification from the server (not the client). This conflict incident is recorded by FileWatcher.

4. The shared folder is updated by the server. The server updates the shared client folder with the most recent version. This guarantees that every user sees the most recent version of the file.

Sync New Files Automatically

1. When the server finds a new file, a fresh file, maybe submitted by a different client, shows up on the server. The file is pushed to clients by the server.
2. This new file is added by the server to the shared client folder. This new file is detected by FileWatcher. The sync event is recorded by FileWatcher with a timestamp.

Implementation

Server configurations

In this phase, socket programming of python was used to setup the Server-client connection. The server would listen to 0.0.0.0 (all network interfaces) at port 65432. It also creates a directory named "server_files". The server continuously accepts connections from clients. For each client, it processes their request. For future, a more secure version of port and host will be used.

The client provides the file name, timestamp, and commands like UPLOAD or DOWNLOAD. To make a decision, the server looks at the timestamp of the file:

UPLOAD: The server accepts and stores the client's file if it is more recent than the server's version.

DOWNLOAD: The server transfers the file if it is more recent than the client.

It sends the proper error messages and gently handles requests that are malformed or commands that are not supported.

The server safely shuts down if it is interrupted (for example, by pressing Ctrl+C).

The server essentially prevents older files from being overwritten while synchronizing files between itself and clients. It resembles a small, personalized file-sharing program! If you would want more information, please let me know.

Client configurations

The client establishes a connection with the server on port 65432, which is running on 127.0.0.1 (localhost). To store files for synchronization, it generates the client_files directory.

Commands (download or upload) are read by the client from the command line. Depending on the directive, it either if the client's version is more recent, it uploads a file to the server or if the server version is more recent, downloads a file from the server.

While uploading a file sends the file name, client timestamp, and command (UPLOAD) to the server. In order to prevent conflicts, the server verifies timestamps. In the event that the server replies "OK," the file is uploaded. The client prints the dispute details if there is one.

While downloading the file, it provides the client's timestamp along with the file name in a command (DOWNLOAD). The client saves the file after the server transmits it if it has a more recent version. The server replies "SKIP" if the client's file is already current.

For syncing every file, the sync_all method attempts to upload and download every file in client_files by iterating over them all.

To prevent overwriting of more recent versions, the client makes sure files are synchronized with the server while taking timestamps into account. It smoothly manages unforeseen reactions or connectivity problems.

FileWatcher configuration

The file watcher program that keeps an eye on a directory to see if anything has changed, particularly when files are created or altered. These new or modified files are automatically uploaded to the server.

The program observes modifications to the client_files directory using the watchdog library. It creates the directory if it doesn't already exist.

For managing events:

on_modified: Initiated upon modification of a file within the directory. After logging the file name and modification time, the revised file is uploaded to the server using the upload_file method.

When a new file is added to the directory, the on_created function is activated. It reports the file details and uploads them, same like on_modified does.

For file upload, it uses the client.py script's upload_file function to call the server and request an upload.

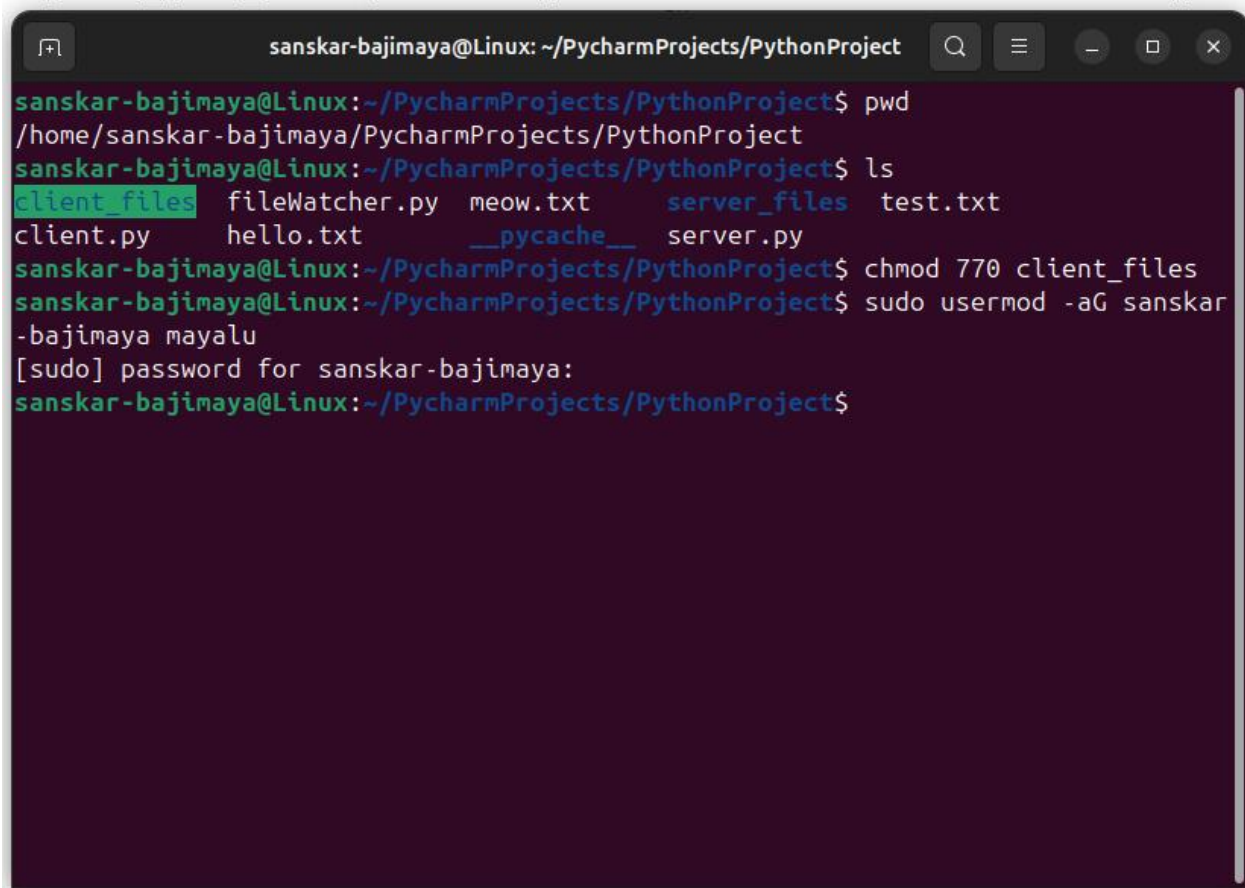
For Process monitoring, the application uses an instance of Observer to begin monitoring the directory. It keeps track of modifications until it is stopped (for example, by pressing Ctrl+C). It politely stops the observer when it is interrupted.

The Principal performance depends upon execution. The script begins monitoring the client_files directory and automatically uploads newly created or modified files to the server.

Goal: This script is good for situations requiring real-time updates because it automatically detects and uploads changes, automating file synchronization.

Results & Explanation

Client_file permission modification

A terminal window titled 'sanskar-bajimaya@Linux: ~/PycharmProjects/PythonProject' showing a series of commands and their outputs. The user first runs 'pwd' to confirm the current directory. Then, they run 'ls' to list the files in the directory, which includes 'client_files', 'fileWatcher.py', 'meow.txt', 'server_files', and 'test.txt'. The 'client_files' directory is highlighted in green. Next, the user runs 'chmod 770 client_files' to change permissions. Finally, they run 'sudo usermod -aG sanskar -bajimaya mayalu' to add the 'mayalu' user to the 'sanskar' group. The terminal shows the password prompt and the successful execution of the command.

```
sanskar-bajimaya@Linux: ~/PycharmProjects/PythonProject
sanskar-bajimaya@Linux:~/PycharmProjects/PythonProject$ pwd
/home/sanskar-bajimaya/PycharmProjects/PythonProject
sanskar-bajimaya@Linux:~/PycharmProjects/PythonProject$ ls
client_files  fileWatcher.py  meow.txt      server_files  test.txt
client.py     hello.txt       __pycache__   server.py
sanskar-bajimaya@Linux:~/PycharmProjects/PythonProject$ chmod 770 client_files
sanskar-bajimaya@Linux:~/PycharmProjects/PythonProject$ sudo usermod -aG sanskar
-bajimaya mayalu
[sudo] password for sanskar-bajimaya:
sanskar-bajimaya@Linux:~/PycharmProjects/PythonProject$
```

In this part of the project, we use the user “sanskar-bajimaya” who created and owns the whole files of fileWatcher, client and server. Also including the client and server files. We change the permission of the client_files as 770 which gives the read, write and execution permission to all the users under the same group as the user “sanskar-bajimaya”.

Then, the user creates a group where “mayalu” user falls inside the sanskar-bajimaya user.

Operating server, client 1, client 2 and FileWatcher

```

sanskar-bajimaya@Linux: ~/PycharmProject...
sanskar-bajimaya@Linux:~/PycharmProjects/PythonProject$ pyt
hon3 server.py
Server listening on 0.0.0.0:65432
Connected by ('127.0.0.1', 59148)
UPLOAD request: Write1
Client timestamp: Mon Apr 7 13:11:05 2025
File does not exist on the server.
File uploaded and updated: Write1
Unsupported command received: UPLOAD
Connected by ('127.0.0.1', 59162)
UPLOAD request: Write1
Client timestamp: Mon Apr 7 13:11:05 2025
Server timestamp: Mon Apr 7 13:11:05 2025
Unsupported command received: UPLOAD
$

sanskar-bajimaya@Linux: ~/PycharmProject...
sanskar-bajimaya@Linux:~/PycharmProjects/PythonProject$ pyt
hon3 fileWatcher.py
Watching: client_files
New file detected: Write1 at Mon Apr 7 13:11:05 2025
Preparing to upload: Write1
Local timestamp: Mon Apr 7 13:11:05 2025
File uploaded successfully: Write1
Modified file detected: Write1 at Mon Apr 7 13:11:05 2025
Preparing to upload: Write1
Local timestamp: Mon Apr 7 13:11:05 2025
Conflict detected: CONFLICT|Server has newer versionERROR|U
nsupported command
$

sanskar-bajimaya@Linux:~/PycharmProjects/PythonProject$ cd
client_files
sanskar-bajimaya@Linux:~/PycharmProjects/PythonProject/client_files$ touch Write1
sanskar-bajimaya@Linux:~/PycharmProjects/PythonProject/client_files$

sanskar-bajimaya@Linux:~/PycharmProject...
cat: /home/sanskar-bajimaya/PycharmProjects/PythonProject/c
lient_files: Is a directory
$ pwd
/home/sanskar-bajimaya/PycharmProjects/PythonProject
$ ls
client_files  hello.txt  server_files
client.py     meow.txt  server.py
fileWatcher.py __pycache__ test.txt
$ cd client_files
$ ls
lover          newFile      Write1
love.txt       program1
'meow.txtMeow'$'\n''Meow mEOW'$'\n'  test.txt
$

```

Here, the top left is the server running. It has all the updates with the updates and modification on the client and server files. The top right is the fileWatcher running. It has all the logs of updates and modifications on the client_files. We can also see that when one file is uploaded on the server from client, it mentions “Conflict detected” and does not overloads the same file at same time more than one time, preventing them from going into infinite loop.

On the bottom left is the client “sanskar-bajimaya”. Notice that when the client creates the file Write1, the file is automatically uploaded to server files and notifies server and fileWatcher’s log.

On the bottom right is the client “mayalu”. Notice that the file created by sanskar-bajimaya is already in the client_files.

```

sanskar-bajimaya@Linux: ~/PycharmProject...
File uploaded and updated: Write1
Unsupported command received: UPLOAD
Connected by ('127.0.0.1', 42042)
UPLOAD request: Write1
Client timestamp: Mon Apr 7 13:11:57 2025
Server timestamp: Mon Apr 7 13:11:48 2025
File uploaded and updated: Write1
Unsupported command received: UPLOAD
Connected by ('127.0.0.1', 43446)
UPLOAD request: Write1
Client timestamp: Mon Apr 7 13:12:03 2025
Server timestamp: Mon Apr 7 13:11:57 2025
File uploaded and updated: Write1
Unsupported command received: UPLOAD

sanskar-bajimaya@Linux: ~/PycharmProject...
Conflict detected: CONFLICT|Server has newer versionERROR|U
nsupported command
Modified file detected: Write1 at Mon Apr 7 13:11:48 2025
Preparing to upload: Write1
Local timestamp: Mon Apr 7 13:11:48 2025
File uploaded successfully: Write1
Modified file detected: Write1 at Mon Apr 7 13:11:57 2025
Preparing to upload: Write1
Local timestamp: Mon Apr 7 13:11:57 2025
File uploaded successfully: Write1
Modified file detected: Write1 at Mon Apr 7 13:12:03 2025
Preparing to upload: Write1
Local timestamp: Mon Apr 7 13:12:03 2025
File uploaded successfully: Write1

sanskar-bajimaya@Linux: ~/PycharmProjects/PythonProject$ cd /home/sanskar-bajimaya/PycharmProjects/PythonProject
client_files $ ls
sanskar-bajimaya@Linux: ~/PycharmProjects/PythonProject/client_files client_files hello.txt server_files
nt_files$ touch Write1 client.py meow.txt server.py
sanskar-bajimaya@Linux: ~/PycharmProjects/PythonProject/client_files fileWatcher.py __pycache__ test.txt
nt_files$ cat Write1 $ cd client_files
I love the way you look at me $ ls
Don't leave me lover newFile Write1
sanskar-bajimaya@Linux: ~/PycharmProjects/PythonProject/client_files love.txt love~ program1
nt_files$ 'meow.txtMeow'$'\n''Meow mEOW'$'\n' test.txt
$ cat>Write1
I love the way you look at me
Don't leave me
^C
$

```

On the bottom right, notice that the file is modified by “mayalu” by writing some things. This automatically triggers the server and fileWatcher to log the modification with time stamp and upload the file to server files. Then on the bottom left, notice that the changes made by “mayalu” user can also be seen in the file of “sanskar-bajimaya”.

This shows that both can work on the same file. However, the different version of the same file can be seen in the server files below.

```

sanskar-bajimaya@Linux: ~/PycharmProjects/PythonProject/server_files
sanskar-bajimaya@Linux: ~/PycharmProjects/PythonProject$ cd server_files
sanskar-bajimaya@Linux: ~/PycharmProjects/PythonProject/server_files$ ls
'hello.txtI love love love you'$'\n' newFile
lover program1
lover~ program1~
love.txt test.txt
love.txt~ Write1
'meow.txtMeow'$'\n''Meow mEOW'$'\n'
sanskar-bajimaya@Linux: ~/PycharmProjects/PythonProject/server_files$

```

We can see that the lover file has two versions. The second version is the older version labeled as “lover~”. A file that ends with ~ in Linux is usually a backup file made with a text editor such as Emacs or Vim. It is a copy of the original file prior to changes being performed, not a temporary

file. If these files are no longer needed, you can manually remove them as they are not automatically erased. These versions of files are all stored in `sever_files`.

Future Work

Delta Sync (Optimization)

The objective is to transfer only modified portions of files in order to minimize bandwidth utilization.

For chunking, calculate checksums (SHA-1/MD5) and divide files into fixed-size blocks (e.g., 1KB). Delta Comparison to find modified chunks, compare the checksums of the client and server.

Sending just modified portions (based on rsync). This affects for large files as the sync time can be reduced by 50–70%.

End-to-end encryption.

The objective is to strengthen security for private information. The plan of implementation is that before uploading, encrypt files client-side using security like AES-256. For key management, storing private keys locally and exchange keys securely using RSA.

Zero-Knowledge Model: Without client keys, the server is unable to decrypt files.

Anticipated Effect: adherence to organizational security standards and GDPR.

GUI for non-technical users.

Objective includes using a graphical user interface to streamline interaction. Plan of Implementation is using PyQt/Tkinter which is a desktop graphical user interface that displays conflicts, version history, and sync status.

Implementing a notification system that inform users of sync events, such as "User Y updated file 'X'."

Conclusion

This project successfully used a client-server model over TCP/IP to implement a distributed file synchronization system based on Linux and influenced by commercial tools such as Dropbox. Among the major accomplishments are:

Real-Time Sync

Time stamps guarantee version control as clients automatically identify and send file changes to a central server.

Conflict Resolution

A "last-modified wins" strategy prevented data loss during concurrent edits, while the server maintained historical versions.

Security

File permissions (chmod 770) and group-based access restricted unauthorized modifications.

Automation

The watchdog library enabled seamless monitoring and synchronization of file changes.

Limitations

Limitations were noted as areas that needed development, including the absence of mobile support and delta sync. Future research will focus on improving security (end-to-end encryption), increasing accessibility (mobile/GUI support), and optimizing bandwidth (via delta sync).

This research shows that it is possible to create a self-hosted, lightweight sync solution that incorporates the essential features of commercial solutions. In contexts with limited resources, it acts as a basis for scalable, safe, and intuitive file synchronization technologies.