CSE 344 – Final Report

MD Sarwar Hossain

The program starts with a sever directory name which will contain multiple client directory. Each client will be connected mapped to those directories. This is important because the directory name is the only security feature in this basic dropbox implementation. So if a client is starts a directory to sync, he/she will not get update of the other directories of the server. This was done deliberately because as a client one is only interested in syncing with a directory he/she wants. Definitely I don't want to upload something and everyone will be able to see it. So the directory works kind of a security measurement here.

SERVER PROGRAM

The server program is designed utilizing the Producer-Consumer design pattern, which is a concurrency design pattern where one part of the application (producer) produces data and another part (consumer) uses it. This pattern is well suited to situations where the rate of data production and the rate of data processing need not be tied to one another.

In this server program, the main thread of the application serves as the producer. It accepts incoming client connections and enqueues them into a client queue. Meanwhile, a pool of worker threads, created during server initialization, serve as consumers. These threads remain idle until a client connection is enqueued, at which point they dequeue the client connection and handle the client's requests.

Synchronization and Thread-Safety

To ensure thread-safety and avoid race conditions, mutex locks (critrgn_mutex, cq_mutex, ft_mutex) are used. These locks prevent multiple threads from accessing shared data simultaneously, thereby preventing inconsistencies.

Condition variables (cq_empty, cq_full) are used in conjunction with the mutex locks to allow threads to sleep until a particular condition is met. In this case, the worker threads wait until the client queue is not empty (meaning a client connection is available), and the main thread waits until the client queue is not full (meaning it can accept more client connections).

The server also employs a signal handling thread to deal with interrupt signals (SIGINT, SIGTERM). This allows the server to gracefully shut down when it receives a termination signal, ensuring that all resources are properly cleaned up.

Client Handling and File Synchronization
The server employs a series of functions to handle client requests and manage file synchronization.

Upon receiving a client connection, the handle_client() function is invoked. This function initiates communication with the client, beginning by sending a connection acknowledgment. The function then proceeds to receive the client's root directory and assess the need for synchronization.

If synchronization is needed, the server sends the client a synchronization start signal and proceeds to send the directory information, followed by the actual files. Once all files have been sent, a synchronization done signal is sent to the client. If no synchronization is required, a synchronization done signal is immediately sent to the client.

To handle file reception from the client, the server utilizes receive_dir_from_client() and receive_files_from_client() functions. These functions facilitate the retrieval of directory and file data from the client and integrate them into the server's file system.

Continuous Synchronization
Following initial synchronization, the server enters a loop where it continuously waits for data from the client and checks for changes in its own directory. If changes are detected or if data is received from the client, appropriate actions are taken to ensure both systems remain synchronized.

Overall, the server program employs a sound design pattern suitable for the nature of the task at hand, which is to handle multiple client connections and facilitate efficient file synchronization. The use of threads and synchronization

mechanisms ensures that the server can handle simultaneous client connections without compromising data integrity or causing resource conflicts.

IMPLEMENTATION

1. Signal Handling: The server uses POSIX signal handling to gracefully handle `SIGINT` and `SIGTERM` signals. The signal handling is carried out in a separate thread `signal_thread` that waits for these signals. Upon receiving a signal, the thread changes the `signal_flag` to `1` and wakes up any threads blocked on condition variables, enabling a graceful shutdown of the server.

2. Thread Pool: The server uses a pool of worker threads to handle client requests. The number of threads in the pool is supplied as a command-line argument and the threads are created during the server initialization in the `main` function. Each worker thread is responsible for handling client requests in a separate function `worker_function`.

3. Client Queue: The server maintains a queue of connected clients, represented by their socket file descriptors. The queue is implemented as a circular array, allowing for efficient enqueueing and dequeueing operations. Synchronization on the queue is achieved using a mutex (`cq_mutex`) and two condition variables (`cq_empty` and `cq_full`).

4. File and Directory Handling:The server includes a range of functions for handling files and directories. The `send_dir_to_client` and `receive_dir_from_client` functions handle sending and receiving directories, respectively. The `send_files_to_client` and `receive_files_from_client` functions handle sending and receiving files, respectively.

5. Socket Programming: The server uses the TCP/IP protocol for communication between the server and clients. The server socket is created, bound to a specified

IP address and port number, and then set to listen for incoming client connections. An accepting loop in the `main` function accepts incoming connections and adds the clients to the client queue. The use of `setsockopt` with the `SO_RCVTIMEO` and `SO_SNDTIMEO` options sets a timeout on the socket's receive and send operations, respectively, allowing the server to periodically check the `signal_flag` and shut down if a signal has been received.

6. Directory Creation: Upon server initialization, a server directory is created based on the command-line argument provided. This directory is used for storing files and directories received from clients. The use of `mkdir` and `chdir` system calls ensure directory creation and moving to the created directory, respectively.

7. Resource Cleanup: The server ensures proper cleanup of resources when shutting down. This includes joining all worker threads, destroying all mutexes, and closing the server socket.

8. Logging: Although explicit logging is not part of the provided code, the server does print out various status and error messages that can be redirected to a log file, if needed. These messages include signal reception, client connection, and server shutdown messages.

Overall, the server is designed for scalability, with its multi-threading model allowing it to handle multiple client connections and requests concurrently. Its architecture is centered around sockets for communication, threads for concurrency, and a queue for managing client connections.

Client Program Design Pattern

The client program for file synchronization follows a common networking pattern often used in client-server architectures. It employs a synchronous communication over TCP/IP where a client establishes a connection with a server and then communicates through sending and receiving data. The program makes extensive use of socket programming to facilitate this.

At a high level, the client program follows a procedural design pattern, which is typical of C programs. This is a straightforward pattern where data and state are kept in global variables, and tasks are carried out by calling a sequence of functions that operate on that data.

This program also employs a signal handling mechanism to ensure graceful shutdown of the client during unexpected termination events. It uses a global `quit` flag to keep track of the signal and a signal handler function to set this flag when a SIGINT (Ctrl+C) event is detected. This allows the program to handle the termination request in a controlled manner, ensuring any cleanup operations can be completed before the program exits.

Individual Functionality

1. Signal Handling:The signal handler `sigint_handler` sets the `quit` flag when a termination signal is received. This allows the main loop in the client program to terminate gracefully, ensuring that all necessary cleanup procedures are performed before the program is shut down.

2.Client Setup:

The main function starts by setting up signal handling, then proceeds to extract the program arguments and initialize the necessary structures. If no server IP address is provided, the program defaults to "127.0.0.1" (localhost). It also sets up the client directory, creates the necessary logs, and initializes a socket connection to the server.

3.Connection to Server:

Upon successful creation of the socket, the client attempts to connect to the server using the provided IP address and port. The program waits for an acknowledgment from the server, informing that the connection has been established.

4.Server-to-Client Sync

The client sends the root directory path to the server to initiate the synchronization. If the server directory is empty, the synchronization is marked as done. However, if there are files in the server directory that are not in the client directory, the server starts sending these files. The client receives these files and updates its directory.

5. Client-to-Server Sync

The client also checks its own directory for any files not present on the server. If there are such files, the client sends these files to the server, synchronizing the server's directory with the client's.

6. Monitoring for Changes and Continued Synchronization:

After the initial synchronization, the client program enters a loop where it continually waits for any updates from the server and inspects its own directory for changes. Any new files or changes in the client directory are sent to the server, ensuring the server stays in sync. Any updates from the server are also integrated into the client's directory.

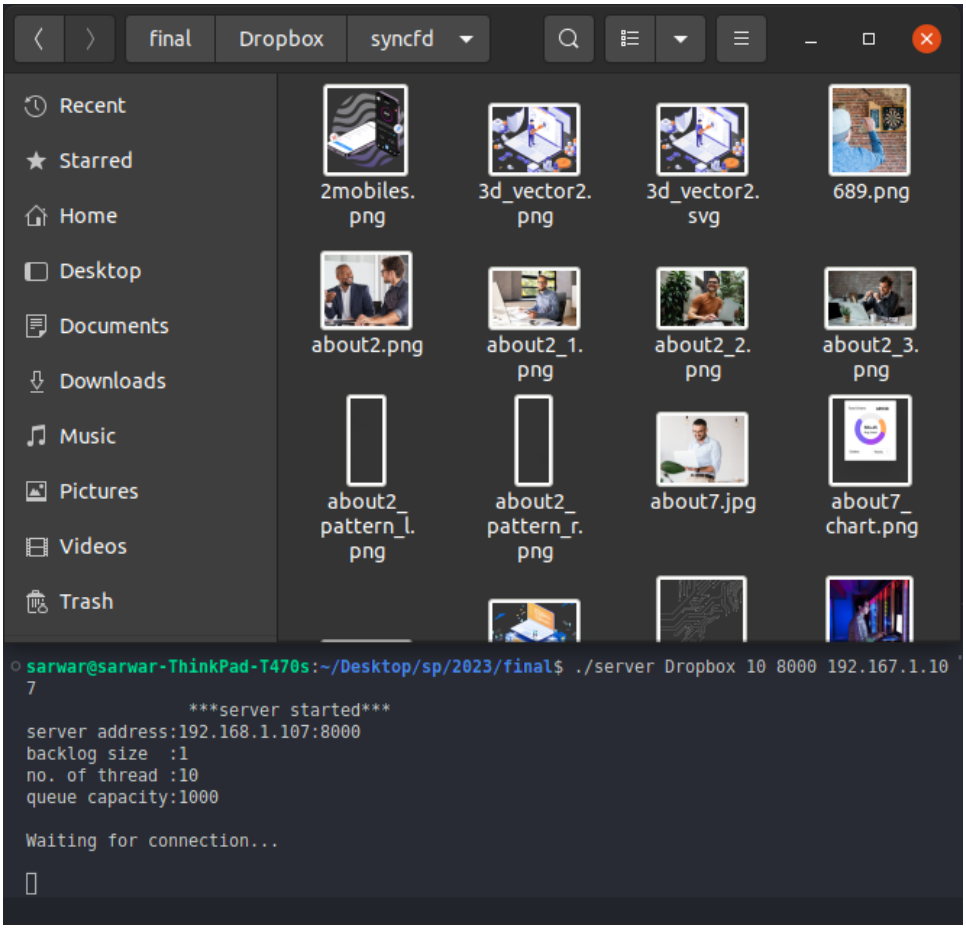
In conclusion, this client program serves as a synchronization agent, maintaining consistency between its local directory and a remote server directory. It utilizes a client-server networking model and procedural programming design pattern to

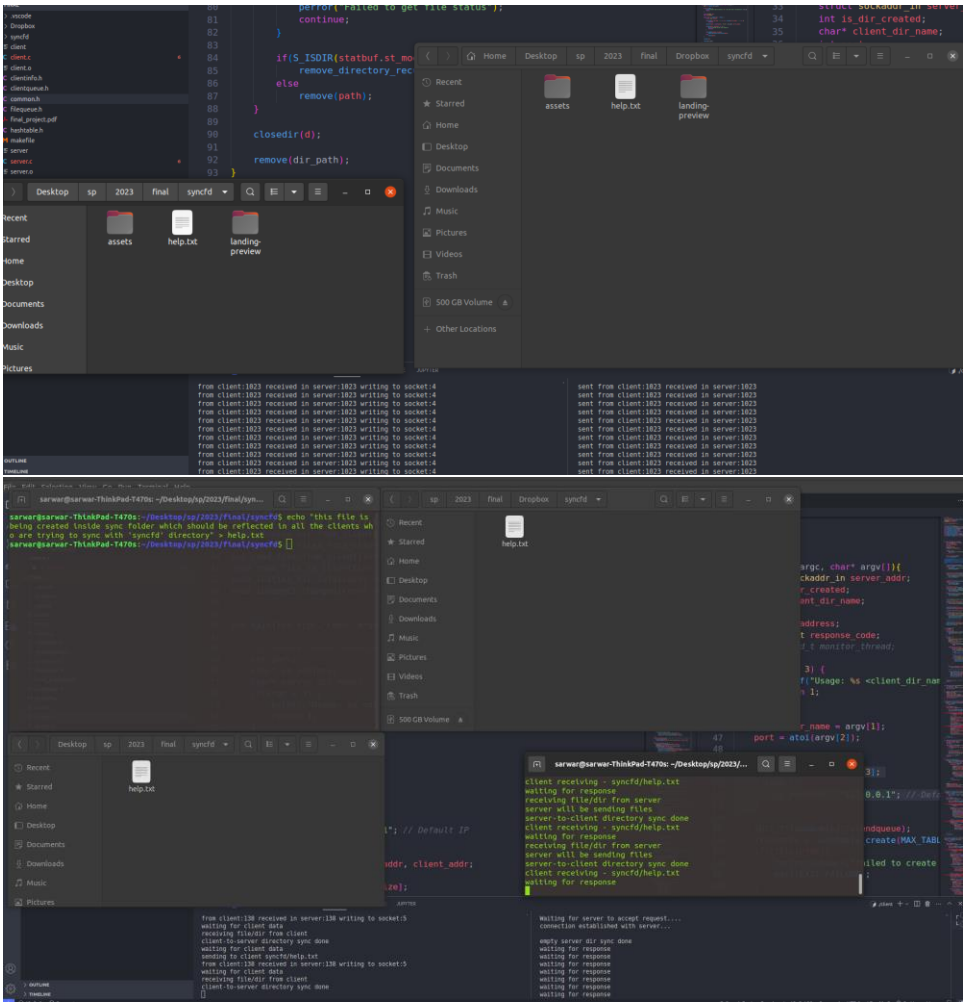
facilitate this functionality. Through the use of various functions for sending, receiving, and handling files, it provides a robust system for real-time file synchronization.

Drawbacks:

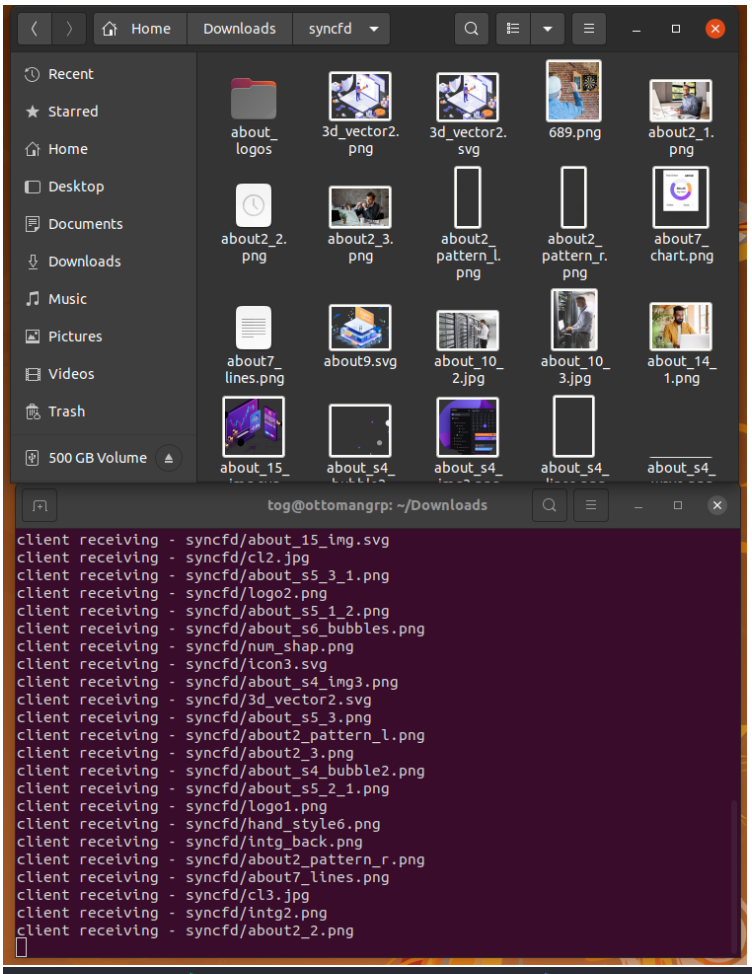
The program don’t have delete operation and file transfer is takes time because even for a small change the whole file is transferred. Should have used checksum algorithm just like dropbox. Sometimes it feels like the program is stacked but it’s just the data transfer over the internet is slow.

PROGRAM SNAPSHOTS:



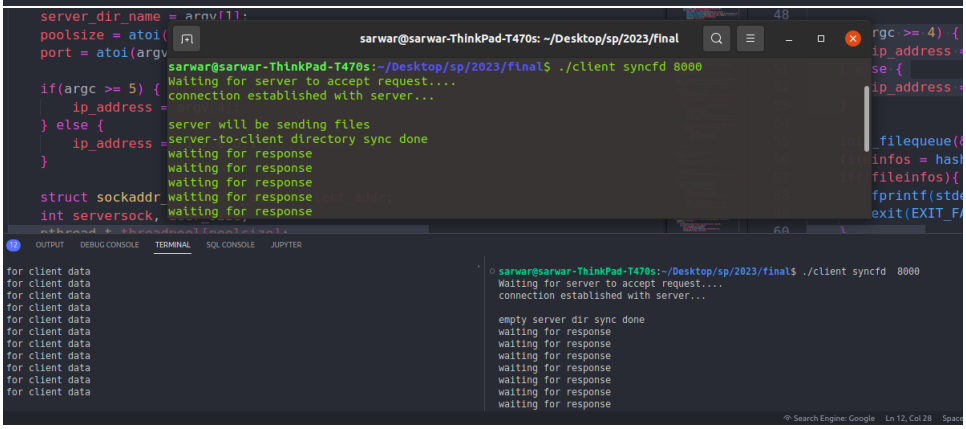

Home
Desktop
sp
2023
final
Dropbox
syncfd
Recent
Starred
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
Trash
500 GB Volume
Other Locations
assets
help.txt
landing-preview
from client:1023 received in server:1023 writing to socket:4
sent from client:1023 received in server:1023

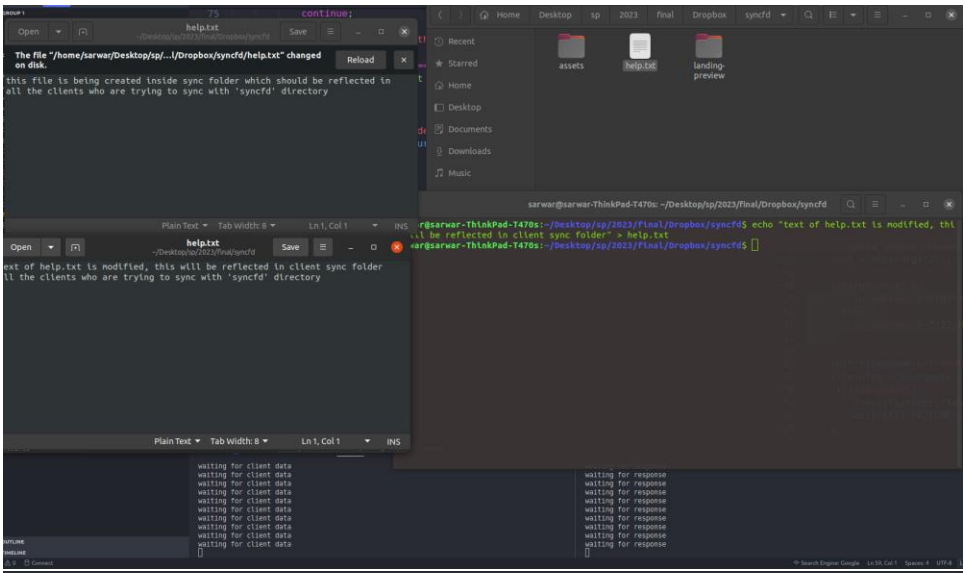
sarwar@sarwar-ThinkPad-T470s: ~/Desktop/sp/2023/final/syncfd$ echo "this file is being created inside sync folder which should be reflected in all the clients who are trying to sync with 'syncfd' directory" > help.txt
sarwar@sarwar-ThinkPad-T470s: ~/Desktop/sp/2023/final/syncfd$
help.txt
client receiving - syncfd/help.txt
waiting for response
receiving file/dir from server
server will be sending files
server-to-client directory sync done
client receiving - syncfd/help.txt
waiting for response




```
sarwar@sarwar-ThinkPad-T470s:~/Desktop/sp/2023/final$ ./server Dropbox 10 8000
                ***server started***
server address:192.168.1.107:8000
backlog size  :1
no. of thread :10
queue capacity:1000

Waiting for connection...
```






```
sarwar@sarwar-ThinkPad-T470s:~/Desktop/sp/2023/final$ ./client syncfd  8000
Waiting for server to accept request....
connection established with server...

empty server dir sync done
waiting for response
waiting for response
waiting for response
waiting for response
waiting for response
waiting for response
waiting for response
```