CSE344

Final Project

Report

1901042676

Mustafa Mercan

Subject:

In this assignment, we were asked to create a dropBox application using sockets. This project consists of two different stages:
1→ Server Side
2→ Client Side
The server side takes parameters such as dirname, port number, and thread pool size. The thread pool size determines the number of threads to be created. Dirname defines the directory where the server will operate, and finally, the port number represents the port for client connections.
The client side takes parameters such as dirname, port number, and IP number. Dirname defines the directory where the client will operate, while the port number and IP number define the necessary information to connect to the server.

SOLUTIONS APPROACH

I started with the server part first. After receiving the parameters on the server side, I created the necessary threads. These threads actually work specifically for each user. Therefore, the threads need to wait in an infinite loop until a new user arrives.
In the server part, I created a socket and started listening. When a user connected, I used the structures I created (the structures below) to store the user's information in a buffer area. As soon as a user is added to the buffer area, the waiting threads start working one by one. From this point on, each thread performs its operations exclusively for the client

```
typedef struct {
    ClientCommand clientCommand;
    int sockfd;
    int thread_pool_size;
    struct sockaddr_in client_address;
    struct RequestQueueNode* next;
} RequestQueueNode;


typedef struct {
    int current_client;
    int count;
    FileInfo *fileInfo;
    char file_path[1000];
    RequestQueueNode* front;
    RequestQueueNode* rear;
    pthread_mutex_t mutex;
    pthread_cond_t empty;
} RequestQueue;
```

When a new client connects, I send the file_path associated with that client from the client side to the server side. Similarly, I also send the file_path of the server's working directory to the client. The reason for this is to enable necessary adjustments when data related to files is transmitted to the other side.
In addition to this operation, if more clients attempt to connect than the thread_pool_size, a message is sent to the other side, and the program enters a waiting state

When the server starts running, it first scans all the files in the directory and stores them in a linked list data structure. This allows me to update all the files here and perform operations based on this data structure.

```
typedef struct{
    int count;
    char file_path[1000];
    FileInfoNode *root;
    pthread_mutex_t mutex;
    pthread_cond_t empty;
}FileInfo;
```

When a new user establishes a connection for the first time, the initial operation is to transfer all the existing folders and files from the server to the client. This process is accomplished using the init_other_side() function. While this is happening within the thread, the init_server_side() function is simultaneously executed on the client side. Since both functions involve the use of send and recv, synchronization is achieved. Whenever a send is used from the server, an incoming message from the client is expected in the same manner. This ensures that the operations between the two sides do not collide with each other.

When the data from the server side is successfully transferred to the client side, the files in the client's directory are scanned and stored in a created linked list. Subsequently, the client enters a loop to listen to the server side. Meanwhile, on the server side, the current directory is traversed to check for any changes. If any modifications occur, such as adding or modifying a file, the corresponding information is updated in the data stored within the linked list. Afterwards, the files in the server-side directory are compared again with the data present in the linked list. At the end of this comparison, if a data entry exists in the linked list but is not found within the server's directory, it indicates that the file has been deleted, and the "deleted_or_not" parameter within the linked list is modified accordingly.

After all these updates are performed within the server, an iterator is created within the generated linked list. While iterating, the flags present in the FileInfoNode structure we created are checked. Some of these flags are "added_or_not," "delete_or_not," and "modified_flag." They store information about whether the corresponding file has undergone any changes.

This mentioned process takes place within the function called update_other_side(). If there are any changes, the FileInfoNode structure corresponding to that file is sent to the client side. The listen_server_side() function running on the client side receives this data and evaluates the flag within it. Subsequently, if it is a newly created folder, it creates the folder. If it is a deleted folder or file, it deletes it from the current working directory. However, if the sent data represents a modified or added file, the client sends a message to the server in the form of "file content request."

On the server side, this message is received and processed. This message indicates the following:
"A file has been created or modified. Therefore, there is a need for the data associated with that file."

Upon receiving this message on the server side, the corresponding FileInfoNode for the required file is located, and the data reading process from that file begins. The data is sent using a structure called sendDataInfoNode. This structure contains the file's path, the path for reading the file's content, and a flag called "done." This flag indicates that all the data has been successfully sent, allowing the client-side loop to break and continue with the processing.

After all these operations are successfully completed, the necessary actions are performed on the modified, added, and deleted nodes within the linked list, and they are cleaned up.

After all these processes, the server starts listening to the client side. In this step, the operations previously performed on the client side are carried out on the server side, and vice versa. This way, file transfers are successfully executed between the server and the client.

The mutexes present within the created nodes help eliminate synchronization issues among the parallel threads running on the server side. These mutexes ensure that the threads operate in a synchronized manner, preventing conflicts and maintaining proper coordination between them.