Kaniel Vicencio and Mathew Alangadan

Architecture Overview for UFmyMusic

Goal

Create a peer to peer file sharing service that allows multiple users to connect to a server and request flies that they do not currently have.

System Components

1. **Client**
   - The client is responsible for sending specific requests to the server and interacting with the local file directory. It supports four primary commands:
     - **LIST**: Request a list of files from the server.
     - **DIFF**: Compare the files on the client with those on the server.
     - **PULL**: Request files from the server that are missing on the client.
     - **LEAVE**: Gracefully end the session and close the connection.
   - The client has a command line interface to enable easy usage of the functions. Where the client inputs a integer 1-4 which will correlate to one of the functions.
2. **Server**
   - The server is the central node responsible for handling requests from multiple clients concurrently. It performs the following tasks:

   Multithreading/Concurrency: The server uses pthreads for concurrency management because our application is intended for a small user base Pthreads offer a simpler and faster solution in this context, as each thread can handle an individual client without the overhead of managing multiple connections in a single event loop like with select(). Additionally, since our clients are only reading from the server, there is minimal concerns about race conditions, making pthreads a more efficient choice for this use case.

   File Management: The server creates a unique hash for every file and uses that to discern if the client possesses a specific file. This is to make sure that if the client renames files, the server will not send them again and waste resources. It is also used to recognize if the client has outdated data. For example, if the server updates one of its text files, the client will be able to pull it again and synch up with the servers content.

   Action Logging: The server tracks historical information about each client by placing a log of their activities in a file. The file is a combination of their IP address and the port number that they used to connect.

   Sending Data
   Based on the requested operation the server will send either a vector of all the files it

possesses (LIST), a vector of all the files it possesses but the client does not (DIFF), or the file data itself (PULL). For pull the data is sent in chunks 1024 bytes.

Also note in the event of a client having a file with the same name as a file in the server, the clients file will be overwritten. For example, assume both the client and the server have a file named "file3.txt" but the content is. If the client pulls the file will synch up with the servers version.

Directory Structure

In the main folder there is the architecture overview that you are reading, a script to run the make files and create the executables, and three subdirectories: clientFolder, multipleConnection, and, serverFolder.

clientFolder: This directory contains the client source file, a makefile, and a subdirectory called clientStore that receives the data from the server.

multipleConnection: This directory is used to test concurrency. The build script will but a client executable in this folder, and it also has a clientStore where it can retrieve data from the server.

serverFolder: This directory contains the server.cpp code, a makefile, and the serverStore. The serverStore contains all the files that are available for the client to pull or list.

Message Types (Data Structs)

There are three structs that are used to help with the functionality of the project.

Server Structs:

```
struct File{
   std::string fileName;
   std::string hash;
};


struct ClientData {
   int clientSock;
   std::string clientIdentifier;
};
```

Client Structs:

```
struct MessageRequest {
   uint8_t type;
```

```
        };
```

File: This struct is responsible for grouping unique identifiers with their file names so the server does not send the client redundant information.

ClientData: This struct allows for the client port and ip address to be passed into the handle client function, so that their history can be logged.

MessageRequest: This struct is used to send the initial request for the server to respond.

Client Functions

void fatal_error(const std::string& message)

Function is for debugging purposes mainly. It alerts user of any issues occurring in the code.

void msg_display(int &option)

Displays the user interface so a client can easily access functions

uint8_t decodeType(RequestType type)

Converts enumerated value into uint8_t value to be sent to server.

void compute_hashes_in_directory(const std::string &directoryPath, std::vector<std::string> &hashList)

Iterates through all files in directory (subdirectories not included) and assigns them a unique hash. This will be used for diff and pull, so that the server knows which files to send and which ones not to.

MessageRequest createMessage(RequestType type)

Takes user input and passes it to decodeType for it to be converted to a uint8_t value

std::vector<std::string> getList(std::vector<uint8_t> &sendBuff, std::vector<uint8_t> &recvBuff, int &clientSock)

First sends uint8_t value to server so it can send back a vector of strings that contains the file names it possesses. It receives the vector from the server then prints it out to the client.

std::vector<std::string> getDiff(std::vector<uint8_t> &sendBuff, std::vector<uint8_t> &recvBuff, int &clientSock, std::vector<std::string> &hashList)

First sends uint8_t value to server so it knows that it will be sent a list of hashes that represents the files that client contains. Then it sends the length of the vector, followed length of each string and the string itself so the server can receive a list of all the files hashed.

void getFiles(std::vector<uint8_t>& sendBuff, std::vector<uint8_t>& recvBuff, int& clientSock, std::vector<std::string>& hashList,std::string& currentDirectoryPath)

Builds off diff functionality but in the last for loop it contains two extra recv functions. One is to recv the number of bytes in the file. The other is recv is in a while loop to receive all the file contents. After that it writes the files the disk.

int main(int argc, char *argv[])

Main is responsible for establishing initial connection to the server, getting the directory path so the directory can be crawled for files, and taking in user input so it can perform the function that the user desires.

<u>Server Functions</u>

RequestType encodeType(uint8_t type)

Converts uint8_t into a a RequestType enumeration so that the server can process the client command

std::string getCurrentTime()

This is used to get the current time for client activity logging.

void logClientData(const std::string& clientIdentifier, const std::string& message)

This function writes to a file where the name is the ip address of a user and port number that they connected to. It logs what operation was performed and at what time

void fatal_error(const std::string& message)

Function is for debugging purposes mainly. It alerts user of any issues occurring in the code.

std::vector<std::string> serverDifference(const std::vector<File> &fileNameHash, const std::vector<std::string> &receivedStrings)

This will compare two vectors that contain the hashes of the files that the server and client have. It looks for duplicates then removes them from the final vector that is returned called difference. This is used in Diff and Pull to not congest the network with redundant information.

void compute_hashes_in_directory(const std::string &directoryPath, std::vector<File> &fileNameHash)

Works the exact same way as the client. Computes a unique hash for every file in the directory.

void listFiles(std::vector<std::string> &fileList, const std::string &currentDirectoryPath)

Iterates through the directory and appends all file names to a vector to return to the client.

std::vector<std::string> getHashList(std::vector<uint8_t> &sendBuff, std::vector<uint8_t> &recvBuff, int &clientSock)

Receives list of files as their hash identifiers from the client so it can be used in the serverDifference function. It first receives the length of each hash, then the actual hash itself and appends it to a vector.

std::vector<uint8_t> copyFileToBuffer(std::string filePath)

Uses std::ifstream to open the file in binary mode. Then it finds the file size and starts reading the entire file into recvBuffer 1024 bytes at a time. Essentially it reads all the bytes of a file into a buffer and returns it so it can be sent to the client.

void executeCommand(RequestType &type, std::vector<uint8_t> &sendBuff, const std::string &currentDirectoryPath, int &clientSock, std::vector<File> &fileNameHash)

This function is the central hub to all the client based commands.

For list it obtains a vector of the files it has in its directory by calling listFiles, then sends the vector back by first sending the length of the vector, and after that sending the length of each file name in the vector, followed by the actual file name.

For diff it receives a vector of strings that contain the hashes of all the files in the clients directory, then the it crawls the servers directory using compute_hashes_in_directory. After it has both he client and server hashes it computes the set difference using serverDifference.Then it sends the file names back to the client in a vector using the method mentioned above of vector length, the element length, then element.

For pull it implements the same code as diff, but also calls copyFileToBuffer. After copying a files content to a buffer it will send the data over in chunks of 1024 bytes to the client .

void *handleClient(void *arg)

This function is the core of handling multiple clients and their requests at once.

The function first assigns the client socket and port to variables for logging use. After that it obtains the current directory, so that functions in the future can iterate through the directory.

After that it goes into a while loop that allows for reads in input from the client, logs the request, and calls executeCommand so the server can deliver the appropriate data.

Req type is used to understand what command the client wants and is passed into executeCommand so the server gives the appropriate response to the client.

pthread_exit(NULL) is used when the client disconnects so that the resources occupied by the thread can be released.

It also detects when a client has disconnected

int main(int argc, char *argv[])

Main is responsible for setting up the initial TCP connection and creating threads using pthread.

Main does socket creation, binding, listening, then enters a while loop and accepts client connections where it will then use pthread_create() to assign a new thread to the client so multiple users can connect at once. pthread_detach() is used to release resources after the client disconnects.