Robert Judka
CS550
Programming Assignment 1

Design Documentation

Background

This is a very simple Peer-to-Peer (P2P) file sharing system allowing for users to connect to a central indexing sever as peers and share files within their directory with the other peers. The design of this system allows for multiple connections between the central server and peers, which allows peers to interact freely with the network without having to wait for other peer requests.

This system was built in bare C++11 (only standard libraries used) on a Linux subsystem. All communication between peers and the server is done over sockets and concurrent requests is supported utilizing threads.

Implementation

Indexing Server

The *IndexingServer* class servers as an interface for starting the server and handling any requests that are received.

Starting Server

When initializing an *IndexingServer* object, a socket is created and bound to the *localhost* address on port **9999**. This connection is maintained for the entirely of the object's lifetime and is only closed when upon destruction. A global *unordered_map<string, vector<int>> files_index* is also declared which will serve as the mechanism for mapping client ids to filenames.

Running Server

Calling the *run()* function of an *IndexingServer* object will start listening on the server's port for any incoming peer connections. When a connection is established, a new thread will be created for the specific client-server connection, and any more requests made by the connection client will be handled within that thread. The server then begins listening for another incoming connection and that process will continue.

With a connection to a peer established, a server then expects a simple protocol for handling the peer's requests. Initially, the peer sends a client id (expected to be that peer's port) and the server uses that id for any other interacts with the peer. The protocols allow for four operations to be performed by a client:

registry(client_socket_fd, client_id)

In this operation, the server expects the peer to send a filename it owns (at max 256 characters, the Linux limit). If received successfully, the server will either add a new entry to the *files_index* (if no other peer has this file) or append the peer's client id to the vector (if other peers have the file).

deregistry(int client_socket_fd, int client_id)

This operation is basically the opposite of *registry()*, except that it removes a peer's client_id from a mapped filename, and removes the filename from the mapping if that peer was the only one to own that file.

search(int client_socket_fd, int client_id)

This operation is the simplest of them all: a peer sends as filename and the server responds with a generated list of all the client ids which own that file. If a filename is received which is not in the mapping, then an empty string is sent back.

In the even that a peer closes the connection or the connection is lost, the server will close the connection with the peer and remove its client id from all files in the *files_index*.

Peer

The *Peer* class servers as an interface for starting the server, handling any requests that are received, and handling user input.

Starting Peer

When initializing a *Peer* object, a socket is created and bound to the *localhost* address on a random port, which is assigned as the peer's client id. Like the *IndexingServer*, this connection is maintained for the entirely of the object's lifetime and is only closed when upon destruction. A global *vector<pair<string, time_t>> files* is also initialized which create a list of all the files within a peer's directory and the respective last modified times.

Running Peer User Interface

Calling the *run()* function of a *Peer* object will create a thread for handling user input. This thread will connect to the indexing server over the **9999** port and send its client id (server port number) to initiate the protocol. With a successful connection established, a new thread will then be started which polls the peer's directory every five seconds and either *registry* or *deregistry* requests to the indexing server, depending on the last know state of the directory. The user interface then has two main methods for sending requests to the indexing server:

search_request(int server_socket_fd)

Here, the user is prompted for a filename, after which the peer interfaces with the *indexServer*'s *search()* operation and displays the result to the user.

retrieve_request(int server_socket_fd)

A user is first prompted for a remote peer to send the retrieve request to. If a successful connection can be established, the user will then be prompted for a filename. If the chosen remote peer has the requested file, the local peer will create the file with the data sent from the remote peer in its own directory. The new filename will be displayed to the user if a successful file download occurred. If not, an appropriate message will be displayed.

A user will be continuously prompted to create a request until they decide to quit the application (removing them from the network).

Running Peer Server

Calling the *run()* function of a *Peer* object will also create a thread for listening on the peer server's port for any incoming remote peer connections. When a connection is established, a new thread will be created for the specific peer-peer connection, and the request will be handled within that thread. The peer server then begins listening for another incoming connection and that process will continue.

With a connection to a peer established, the peer server receives a filename from the peer client and attempts to locate and read the data in the file within its directory. If the file can be read successfully, the server peer will send the file size to the peer client, after which the peer server will send the contents of the file in 4096-byte blocks until the file is completely trasnfered. This process is invoked by the peer server's *retrieve(int client_socket_fd)* operation.

Tradeoffs

No Subdirectories

The design specifically ignores any subdirectories as I feel like it was not in the scope of this project. With subdirectories, it would not be safe to assume that a peer's filename is at max 256 characters and thus would require additional communication between the peer and indexing server to identify that size. It would also require a protocol for deciding how mapping should occur for filenames that are the same but are in different subdirectories.

Removing Modified Files

Anytime a file is modified, a *deregistry* request is sent to the indexing server to remove the file. This decision was initially made to reset the modified time of a file in the *files_index* map, however after discussing with one of the TAs, it was decided that this functionality would make the system too complex.

Renaming Downloaded Files if File Exists in Directory

As an example, if peer1 has file 'a.txt' and wants to download peer2's 'a.txt', peer1 will end up with a file called 'a-origin-{peer2's client id}.txt'. Since the peer could not assume if the user wanted to override, append, or create a copy, the system will simply append the origin of the file, creating a new version of the file. This also resolves the issue of needing to know the contents of a file upon download.

5 Second Automatic Update Interval

It was chosen that a peer would send requests to update the *files_index* every 5 seconds instead of continuously. This was done to minimize the number of requests being sent to the indexing server. Because of this five second delay, however, a peer client may be able to see a peer mapped to a specific file even though it does not own the file. If the peer client requests that file, they will then receive a failed message instead of the file they expected.

Immediate Disconnect on Error

A server will immediately disconnect from a peer if any issues occur during communication. The purpose of this is to not interfere with other connections or cause total network failure (for the indexing server). In the instance a peer is disconnected from the indexing server, the peer will be removed from the *files_index* as it is no longer accessible.

Improvements and Extensions

NoSQL indexing server *files_index*

By utilizing a NoSQL database solution for the indexing server, we could provide faster searches (especially for larger number of peers and files) and backup solutions in the even the indexing server fails.

Subdirectory Support

Allowing subdirectories to be indexed by the indexing server would require the indexing server have knowledge of more complex paths and being able to recursively send all the files within a directory.

Optimal File Transfers

Instead of prompting the user which peer they want to download a file from, the indexing server could determine which peer to download from by maintaining some state of open connections. The indexing server could also facilitate splitting up the file transfer into chunks to distribute a download from multiple peers.

Server Class

For better maintainability, a *Server* class could be created to decrease duplicated code across the *IndexingServer* and *Peer* classes. By having both these classes inherit from the *Server* class, the *Peer* class can have the necessary functions for starting and running its server.

Failure Retires

Right now, any failures result in the disconnection from the server. By implementing some sort of retry mechanism, the server could attempt to recover the connection by trying to recommunicate with the peer to continue the protocol. It would also be beneficial to include some retry limit to prevent a server from wasting its resources with a difficult peer.