Robert Judka
CS550
Programming Assignment 2

Design Documentation

Background

This is an extension of the simple Peer-to-Peer (P2P) file sharing system from PA1 to support a hierarchical structure, removing the central component. To accomplish this, we will be creating a network with peers (which act as indexing servers) and nodes. The design of this system allows for multiple connections between the peers and nodes, which allows nodes to interact freely with the network without having to wait for other node requests.

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

Configuration

In this implementation, a configuration file is used to establish the overall network topology and assign some static values. There is a single configuration file for peers and nodes, with the syntax for each differing slightly. The first line in the configuration file defines the TTL value to be used for the network. Then, the following lines are used to add both peers and nodes (in that order). The syntax for that is defined below.

Super Peer

An example of defining a peer in the configuration file is

0 1 55001 55000,55002,55003,55004,55005,55006,55007,55008,55009 55011,55012

Here, the 0 is the key for describing this as a peer, the 1 is the id associated with this peer, the next number is the port to be used for this peer, the next set of comma-separated numbers are the neighbors for this peer, and the next set of comma-separated numbers are the nodes associated with this peer.

Leaf Node

An example of defining a node in the configuration file is

1 0 55010 55000

Here, the 1 is the key for describing this as a node, the 0 is the id associated with this node, the next number is the port to be used for this node, and the last number is the peer associated with this node.

This structure allows any peer or node to be run by simply proving an id to match it with the connections it must maintain.

Implementation

Super Peer

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

Starting Super Peer

When initializing a *SuperPeer* object, a socket is created and bound to the *localhost* address on the port assigned in the configuration file. This connection is maintained for the entirety of the object's lifetime and is only closed upon destruction. A global *unordered_map<string, vector<int>> _files_index* is also declared which will serve as the mechanism for mapping node ids to filenames.

Running Super Peer

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

With a connection established, a server then expects a simple protocol for handling the requests. Initially, the connected device sends an identity tag to establish itself as either a node or a neighbor peer.

For a node connection, it first sends an id (expected to be that node's port) and the server uses that id for any other interacts with the node. The protocols allow for three operations to be performed by a node:

registry(int socket_fd, int id)

In this operation, the server expects the node 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 node has this file) or append the node's id to the vector (if other nodes have the file).

deregistry(int socket_fd, int id)

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

node_search(int socket_fd, int id)

Here, a node sends a filename and the server responds with a generated list of all the ids which own that file. To generate that list of ids, the server first queries its own mapping and then sends a query to all of its neighbor peers. The server then appends all the responses from the neighbor peers in the list sent to the node. If a filename is received which is not in its mapping or any of the neighbor peers, an empty string is sent back.

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

For a neighbor peer connection, there is a bit more message passing required as each peer must also manage which messages it has seen. However, there is only one message protocol used between peers. First, the neighbor peer sends a TTL value for the message (to prevent messages from being forwarded an infinite amount of times), the two parts of the message id (in our case the requesting node id and a sequence number) and the filename to search for. The message id then gets checked against the peer's global *_message_ids* mapping to see if the request has already been seen and forwarded. If it has been, it simply sends back an empty string. If the peer has not seen the message before, it checks its own mapping and broadcasts the message to all its peers, decreasing the TTL value. After receiving the message back from all its peers, it passes that message along to the original sender.

Leaf Node

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

Starting Leaf Node

When initializing a *LeafNode* object, a socket is created and bound to the *localhost* address on the port assigned in the configuration file, which is assigned as the node's id. Like the *SuperPeer*, this connection is maintained for the entirety of the object's lifetime and is only closed upon destruction. A global *vector<pair<string, time_t>> _files* is also initialized which create a list of all the files within a node's directory and the respective last modified times.

Running Leaf Node User Interface

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

search_request(int socket_fd)

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

obtain_request()

A user is first prompted for a remote node to send the request to. If a successful connection can be established, the user will then be prompted for a filename. If the chosen remote node has the requested file, the local node will create the file with the data sent from the remote node 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 Leaf Node Server

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

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

Tradeoffs

The same tradeoffs from PA1 were also made here. As a quick summary, those were: no subdirectories, removing modified files, renaming downloaded files if the file exists in the directory, 5-second automatic update interval, and immediate disconnect on error. In addition to the aforementioned, there were specific tradeoffs made for this implementation.

Iterative Broadcasts

When a peer broadcasts a message to all its peer neighbors, it simply (randomly) iterates through its list of neighbors from the configuration file and broadcasts the message one by one. Upon some testing, it was found that threading this process (so that each message was broadcasted concurrently) had more overhead and caused a request from a node to take ~30 times longer.

1 Minute Message Cleanup

To prevent a peer's message mapping from growing infinitely, a thread is run which removes any old messages (older than 1 minute) every minute. These times were decided as it is okay for this mapping to grow large, however, there should still be some maintenance for its size. Also, as the average message time was found to be around .4-.6 seconds, 1 minute should be plenty of time for a message to be transmitted across the entire network.

Improvements and Extensions

The same improvements and extensions from PA1 can also be made here. As a quick summary, those were: subdirectory support, optimal file transfers, *Server* class, and failure retries. There are also other improvements that could be made in this implementation.

Continuous Connections Between Super Peers

In this implementation, we decided that the connections between peers should only be opened on a *need* basis. During development, this provided acceptable results for nodes issuing queries. However, while performing the evaluations, the constant opening and closing of these connections lead to a huge and unexpected degradation in performance. As we are using a configuration file for initializing each peer, we can utilize that the open the connections when a peer is started, and only chose them on any sort of error or when the server closes.

Message Serialization

A massive improvement (both for performance and programmability) would be to serialize the message instead of passing everything as individual messages. To accomplish this, there would need to be some sort of serializing interface which would translate complex objects that both ends of the connection would understand. Then, single messages could be utilized for sending data (such as the message id).