Robert Judka
CS550
Programming Assignment 3

Design Documentation

Background

This is an even further extension of the simple Peer-to-Peer (P2P) file sharing system from PA1 to support a hierarchical structure and various consistency methods. To accomplish this, we will be creating a network with peers and nodes. The design of this system allows for multiple connections between the peers and nodes without waiting for other node's requests.

This system was built in bare C++11 on a Linux subsystem. 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, consistency method, 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 consistency method to use (0 for *PUSH*, 1 for *PULL FROM LEAF NODES*, 2 for *PULL FROM SUPER PEERS*), a TTR value (used only with consistency methods *PULL FROM LEAF NODES* and *PULL FROM SUPER PEERS*), and a 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.

Communication 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 persistent socket is created for any connections to the peer, and a global *unordered_map<string, vector<int>> _files_index* is declared which maps leaf node ids to filenames.

Running Super Peer

When the peer is run, it listens for any incoming 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 listens for any other connections. 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. If received successfully, the server will add the file to its *_files_index*.

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.

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 (from its own mapping and from broadcast messages sent to neighbor peers).

If a node leaves the network, the connection is closed, and its id is removed from *_files_index*.

For a neighbor peer connection, there are three message protocols used. Each first starts by comparing the TTL value and message id. The rest of the protocols are described below:

query(int socket_fd)

The server gets any matched ids from its mapping, then broadcasts the message to any of its neighbor super peers. It then returns a list of all the matched ids.

invalidate(int socket_fd)

This operation is used when the *PUSH* consistency method is enabled. It sends an invalidation message to any of its connected nodes, then broadcasts the message to any of its neighbor peers to invalidate and remove any cached files that have been modified by the origin server.

compare(int socket_fd)

This operation is used when the *PULL FROM SUPER PEERS* consistency method is enabled. It initially sends this message to all its neighbor peers, who then checks their connected leaf nodes for the modified file. If a node has a stale version of a file, an invalidation message is sent.

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 persistent socket is created for any connections to the peer and a global *vector<pair<string, time_t>> _local_files* is initialized of all a node's local files.

Running Leaf Node User Interface

When the node is run, a thread will be created which connects to its peer and initiates the protocol. A new thread will also be started which polls the node's directory for any changes to its local files. 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 and filename to download. If the file is found, it will be downloaded to the node's remote files directory. This interface also functions to refresh a remote file, where if the node already has a matching file it will simply be updated.

A user will be continuously prompted to create a request until they decide to quit the application.

Running Leaf Node Server

Running the node will also create a thread for listening on the node server's port. 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 server will then continue listening for any other connections. Initially, the connected device sends an identity tag to establish itself as either a node or a neighbor peer.

For a node connection, the protocols allow for two operations to be performed by a node:

handle_obtain_request(int socket_fd)

In this operation, the node server checks its stored files (first local, then remote), for the file to download. If the file exists, the server sends the file, along with the origin id (for a remote file) and the version.

handle_poll_request(int socket_fd)

This operation is used when the *PULL FROM LEAF NODES* consistency method is enabled. The server will check its *_local_files* to see if the file and its version match and send back the result of the check.

For a peer connection, there is a single message protocol used for any of the consistency methods. This protocol checks a server's remote file version against the version from the origin node. If the versions are not consistent, the file is removed and marked as valid from the node's remote files list.

Consistency Implementation

To keep track of remote files stored by a leaf node, the leaf node's directory has a "local" and "remote" directory which organize their files. Any remote file maintains its locally saved name, the origin name, the origin node, the version, the last check time, and the file's validity. This allows the remote files list to act independently of the consistency method used, allowing for a lot of the methods to be reused for each method.

PUSH

To achieve file consistency using the *PUSH* method, a super peer's *deregistry* protocol can send an *invalidate* message to all its connected leaf nodes, where each leaf node then checks its remote files list and removes the file if it exists. This message then gets broadcasted to any neighboring super peers following the standard message propagation techniques.

PULL FROM LEAF NODE

To achieve file consistency using this method, a leaf node simply must poll the origin server for its remote files if its TTR value has expired. This is done while the node is registering files with the super peer. If the origin server responds that the version is invalid (or the file no longer exists), then the polling node will simply delete the file and mark it as invalid.

PULL FROM SUPER PEER

To achieve file consistency using this method, a super peer must be able to check with its neighbor super peers every TTR if any cached versions of a modified files are stale. To do this, a super peer starts a thread which,

every TTR, compares the version of a modified file with any of its leaf nodes which have a cached version of that file. The peer then sends that message to all its neighbor super peers who then also compare versions with their respective leaf nodes. The modified files list gets populated with every *deregistry* request sent from a leaf node, which contains a modified file's name, origin id, and new version.

It is to be noted, however, that the leaf nodes lazily maintain their remote files list and their super peer's files index. That is, even though a file has been removed it does not necessarily mean the super peer is aware of it yet (the leaf node only sends updates the super peer every 5 seconds).

Tradeoffs

The same tradeoffs from PA1 and PA2 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, immediate disconnect on error, iterative broadcasts, and 1-minute message cleanup. In addition to the aforementioned, there were specific tradeoffs made for this implementation.

Assume Download from Local Files First, Then from First Found Remote File

Essentially, this boils down to an unspecified technique for managing files with the same name. If a leaf node downloads a file from some other leaf node that has the same name as one of its local files, then it would be seen as having two versions of that file in the super peer. The reason for not renaming the downloaded file is because if any other leaf node wanted to download the cached version, there would be no distinction that the file is the same or not. For the same reason, it is also assumed the first found remote file is the one to download.

Improvements and Extensions

The same improvements and extensions from PA1 and PA2 can also be made here. As a quick summary, those were: subdirectory support, optimal file transfers, *Server* class, failure retries, continuous connections between super peers, and message serialization. There are also other improvements that could be made in this implementation.

Handling Leaf Node Disconnections

Currently, it is assumed the topology is static and each leaf node is connected to some super peer. It would be a very useful extension to handle any leaf nodes that disconnect by removing any cached files from that specific node. To accomplish this, the cleanup method would have to depend on the consistency method assigned in the configuration file, where the node would either broadcast a message to delete all of its cached files, or every other device in the network would have to handle removing their cached version if that specific leaf node is no longer reachable.

Query Shows Origin Server of Cached Files

Related to the tradeoff with downloading files with the same name, a way to circumvent that tradeoff would be for any queries to also include whether the node is the origin server or if it just has a cached version of the file, and if the file is cached version then who is the origin server. This could be implemented in a few ways, but the simplest would probably be to just send 2 query messages, one that contains all the nodes that are considered origin servers of a file, and one that contains a mapping of the node with the cached version and the respective origin server. That could be further simplified by creating 2 separate requests for a node client, one for only seeing the origin servers and another for seeing all the cached owners.