Overview

This is a peer-to-peer (P2P) system with an indexing server (super-peer (SP)) and multiple clients. Multiple super peers can connect together in two different design configurations (broadcast and linear) these are talked about in the topology section below. The function of the indexing server is to provide a single connection point for the client to join the P2P system. The indexing server is at a known address and uses a known port, so whenever the client program starts it reaches out to a static IP/port to check if the server is running. The job of the indexing server is to keep track of which clients are actively participating in the P2P system, maintain a list of files available for download, and store the client contact information for each file. None of the files available for download are stored on the indexing server, instead the index server provides the contact information (IP and port combination for the peer that has the file) of any requested file to a client. The client then uses the contact information and makes a request to another client in the P2P system for the file. This means that each client in the P2P system has the responsibility of maintaining a "FTP mini-server" within its client application. This "FTP mini-server" simply waits for a connection and receives files through that connection. This client FTP server plays a crucial role in maintaining consistency throughout the system by tracking the state of every file on the client machine. The indexing server program doesn't provide any interactive prompt when running, but does log client, file, and request information to stdout in the terminal running the server program. The client program provides a interactive CLI that asks for user input as commands while hiding the register/deregister functions from the user. All communication between clients and to the indexing server use a multi-threaded asynchronous RPC framework. This is explained in more depth in the sections below. The RPC framework used is gRPC, a framework developed at Google that uses protocol buffer syntax, to serialize the input/output of each function.

Server

The high-level responsibilities of the indexing server:

- Provide an entry point for a client to the P2P system
- Maintain a list of all clients using the system
- Organize and store client meta-data
- Maintain an up-to-date list of files available for download
- Remove a client (and their files) when they disconnect from the system

The indexing server program creates a multi-threaded asynchronous server bound to the localhost on a static predetermined port. This address is hard-coded into a configuration file that is read at startup. Once the indexing server program is started it doesn't expect (or accept) any user input. The indexing server uses an async threadpool to handle multiple client requests at a time. The only data structures the indexing server uses is a vector (list) to track the clients and a vector (list) to track all files. A hashmap could have resulted in a faster search time, but only in the event of a perfect string match. A vector (list) was chosen because it would be easier to implement a partial match function in the future. Since this is a multi-threaded program both the client list and file list are wrapped in a mutex to prevent race conditions. The indexing server prints to stdout a running log of events during execution. The indexing server creates a separate query server to handle the broadcasting and processing of queries received by clients.

The RPC functions the indexing server provides for clients are:

- Register – Logs the clients ID, IP, port, state, and files in the appropriate data structures.

- Deregister – Receives a client ID and drops its info from the server's data structures.

- List – Returns a list (of unique values) of all files available for download between all the local clients directly connected to the indexing server. The filtering out of duplicate values is done on the server. This is mainly a convenience function since the search function requires an exact match.

- Search – Receives a file name and returns a list of other clients registered to the indexing server that have that file available for download. This first broadcasts the query to its neighboring super-peers then performs a search on the files locally indexed on the server.

Client

The high-level responsibilities of the client:

- Provide distribution transparency to the end user
- Generate a unique ID
- Create and bind a service that can receive a file from another peer-to-peer
- Be able to send a file to another peer when asked
- Able to auto-update its available files when there is a change
- Provide a list of files available for download

When the client program starts it reaches out to a hard-coded address from a configuration file read on startup to see if the indexing server is running. If the server is running the client sends its metadata to the indexing server and spins up a local FTP server and query server on predetermined ports. The metadata sent to the indexing server includes; IP, port running FTP service, port running the query server, unique ID (hash of current directory and MAC address), files available for download, and current state (a hash of all files available for download). The client has, at a minimum, four threads of execution. These threads are the following:

1. FTP server to download/receive files (This uses an async threadpool to handle requests)
2. Query server
3. User input from terminal
4. Auto-update mechanism and control logic

The first two threads listed above run independently. Whereas, the last two threads communicate with message passing. This introduces a few extra variables and overhead, but eliminates the potential for a race condition when using a shared-memory implementation. Even though the minimum thread count is four for the client, it normally runs 17-20 threads during execution due to the async threadpool. A sleep function is implemented in the the fourth thread listed above to reduce CPU spin-locking.

The RPC functions the client program implements are:

- Implicit register – During program startup the client information is sent to the server without the end user explicitly asking.

- Implicit deregister – The client side structure has a trait that will automatically send a deregister request when the program terminates. The end user doesn't have to call any function.

- Implicit update – A client's state is generated by hashing all the files in its current directory. This hash is recalculated continuously and if there is a change the client program will deregister and then immediately reregister with the indexing server.

- Get – This will first send a search request to the indexing server (which broadcasts it to other SPs) and then sleeps for 300ms to allow all the queries from different SPs to arrive. The client then requests the file from the first client peer in the list via RPC. The RPC call includes the name of the requested file and the IP/port that the client's FTP server is running on. The peer that receives the Get request will send its file to the port the client included in the function's parameters.

- Send – Used when it receives a "Get" request from another peer. The client will send the requested file to the peer on the IP/Port that the peer sent in the "Get" request.

- List – Requests a full list of all files available from the indexing server.

- Refresh – The client will re-download all files marked as invalid from their original source.

Query Server

The high-level responsibilities of the query server:

- Receive queries
- Broadcast queries to neighbors
- Routinely process and clear queries
- Send query results back to the requester

Both the client and index server run a query server. The query server acts as a middleware that handles the broadcasting and management of queries. Like the client code this query server runs in a different thread an periodically (every 20ms) checks, processes, and broadcasts pending queries. Additionally every 40s the query engine will clear its sent_queries list removing all but the most recent entries. This is to prevent memory leak. The typical cycle of the query server is to:

1. Receive query
2. Add to pending query list
3. Broadcast query to neighbor(s)
4. Add to sent query list
5. Process query
6. Return query results to the requester
7. Clear query from pending query list
8. Routinely clear the sent query list

The RPC functions the client program implements are:

- Receive – Adds query to the pending query list
- Broadcast – Broadcasts query to super-peer neighbors
- Process – Pass search request to super-peer and receive results
- Broadcast_response – Return results of a successful broadcast
- Response_read – Read responses from broadcasted query
- Clear – Clear query lists

Consistency

The entire system maintains consistency by using either a push or pull methodology. The methodology used is configurable via a CLI argument passed when the binary is ran, but by default it uses the push method. Both methods work regardless of the SP topology (broadcast or linear).

Push

This is the default consistency method used throughout the system. This method will broadcast out an invalidation message using the query engine discussed in the section above when ever it detects a change in one of the files in the “Owned” folder. The running execution process is:

1. At start up create a checksum of every file in the “Owned” folder
2. Every 100 ms re-compute the checksum
3. If there is a discrepancy broadcast out an invalidation message for that file and store the new checksum
4. The SP and Query Engine deliver the message to every client that has a copy of the invalidated file.
5. The client checks the query queue every 100ms for any messages.
6. The client receives the invalidation message and removes the file from its “Downloaded” folder while storing the IP address for re-download if the user wants to request the file again via the refresh command.

Pull

This method doesn’t use the Query Engine to send invalidation message, but instead relies on the FTP server on each client to periodically check to see if a file has based its expiration date and then re-polls the original server to see if the file is still valid. This method requires the FTP server keep track of every file’s expiration date and checksum (it uses the checksum to compare if there are any changes from the original server). The running execution process is:

1. The client downloads a new file from a peer and receives the TTR, expired time, source ip, and checksum for the file.
2. If the expired time is zero (in the case that it downloaded the file from the owner) then the client takes the current time, adds the TTR, and sets that as the expired time. Otherwise the client sets the expired time as it is sent from the peer.
3. The client then sends all of this state information to the FTP server. The FTP server maintains this state so it can directly re-poll the source servers without chaining RPC from client->ftp->ftp->client
4. Every 100ms the client asks the FTP server to re-poll any expired files.
5. The FTP server computes the current time to find expired files. Then goes through each one and re-polls the original server.
6. If the checksum is the same, the FTP updates the expired time to the current time plus TTR. Otherwise the FTP server marks the file as invalid, deletes the file from the "Downloaded" folder, and notifies the user it can be refreshed.

Limitations/Improvements

The current version of the P2P system only works when both server/client are ran on the same machine. This is because the IP of the server is hardcoded to the local host. Future work would change this to broadcast a request to the local network to find the indexing server.

Currently the server relies on the client to send a deregister request. Although this is an automatic function when the client program is terminated there are still cases that prevent the client from sending the request. Such as when it receives a SIGKILL from the OS. Future work would have the server performing a routine health probe and deregister any client that doesn't respond.

Future work could introduce a caching system and map of nearest peers to reduce indexing server reliance. This could potentially allow a client to run even when it can't reach the indexing server by using another peer as a proxy.

The indexing server is a single point of failure and there is no server-side data persistence or fault tolerance. Error handling is performed at the client to gracefully shutdown the program when the server isn't running, but future work would prevent the server from crashing to begin with.

The indexing server does some server-side processing when the client asks for a full list of all files available for download. The server will filter out any duplicate file names. This could move to the client side in the future to lessen server load.

The query server keeps a list of all its sent query and periodically clears all but the last entry from the list (every 40 seconds). Instead of clearing the list at a set interval this could be replaced with a circular buffer that will overwrite old entries.

The search function is a linear search against a list of all files. This could be improved upon (via hashmap magic or a fancy search algorithm) in the future to reduce runtime.