File Sharing Program

R. van Tonder, 15676633
A. Esterhuizen, 15367940

October 23, 2011

Contents

1 Description

In this project, a peer-to-peer file-sharing program was implemented. It consists of a client/server model, with some constraints. In particular, search requests should be forwarded by the server, and clients cannot transfer files preceding a key check, and request forwarding by the server. Thus, the client front-end gives no indication of where files are downloaded from or uploaded to. The most pertinent exercises of this project included implementing a distributed search, pause/resume functionality, and establishing an upload/download protocol. Other important aspects included use of multiple threads for GUI purposes, key encryption, and search string matching.

2 Features

2.1 Included

2.1.1 Server

The server has the following capabilities, in accordance with the project specification.

1. The server accepts connection requests from clients, and updates the user-list with the username of connected clients.
2. Usernames must be unique, and are checked by the server at connection time.
3. The server is responsible for forwarding search requests in a distributed manner to all other clients.
4. The server is responsible for forwarding download requests to all clients, and "introduces" clients to one another.
5. Maintains connection information of all clients.
6. Whispering and global chat message relaying.
7. A GUI which displays
    - All messages sent through the server
    - Client connections to the server
    - Client disconnections from the server
    - The current user-list

2.1.2 Client

The client has the following capabilities, in accordance with the project specification.

1. Clients may connect and disconnect without incident.

2. Commands that are processed by the server for various functionalities, such as `\search`, `\download`, `\pause`, `\resume`, and `\msg`. See **??** for more.
3. Searches return exact matches as well as close matches.
4. Single upload or download stream.
5. Concurrent upload and download streams.
6. Files download/upload correctly.
7. Pause/Resume functionality.
8. A GUI which displays
    - The current user list
    - All global messages, and whispers when applicable
    - Progress indicators for both uploading and downloading.

2.2 Features Not Included

File list directories are hard-coded in the implementation. GUI look-and-feel, and progress bar colours, are not customizable.

2.3 Extra Features

Encryption has been added to the implementation, affording greater security. Upon receiving a download request, a client will compare their own key to the key received in the request. If the keys match, the client will start uploading the file requested. Sending client keys over the wire can make them susceptible to packet-sniffing. It is conceivable that a malicious person could obtain client keys, and use them to impersonate or otherwise permanently occupy upload slots.

For this reason, keys are first encrypted with an *affine substitution cipher*[1] before being sent over the wire. The server never requires decrypting this key, and simply forwards it. Receiving clients then decrypt the key and compare it to their own. While not the most robust security measure, it will at least deter amateurs who hope to simply make use of packet-sniffing.

3 Design, Data Structures, and Algorithms

3.1 Server

The server is responsible for forwarding messages between clients, download requests, and search requests. File lists are not stored on the server, and for this reason actual searches are performed client-side. The server maintains a dictionary mapping of connected clients and connection hash values. Because the server needs to identify where search requests come from, and who to return

[1] See http://en.wikipedia.org/wiki/Affine_cipher

search results to, some sort of identifier is necessary. However, using the IP of clients is undesirable as clients then receive information that is not intended for them. Thus, the connection hash values are used, which are essentially a hash of the socket object represented by a client on the server. Conversion to, and from, the hash value is done server-side.

3.1.1 Operation of Distributed Search

The server distributes search requests made by clients and are recognized by a `__search` token perpended to a message. After each client has performed a search, it returns its results and originating client hash has to the server, prepended by `++search`. The server uses the dictionary to look up the IP of the client who originally issued the search request. Results of search requests are output on the originating client as they are received via the server. While it would have been fully acceptable to temporarily send client file lists to the server for processing, the implemented search is much more distributed in nature.

3.2 Client

3.2.1 Overview

The client is launched with four command line arguments. They are: `<host-name> <port> <nickname> <receiver-port>`. Upon start-up the client establishes a connection with the server at the specified host-name and port, and requests that its nickname be validated. If the nickname is already in use, the user will be notified on the command line window, at which point the client will exit. The `<receiver-port>` parameter is the port number used by the client for uploading and downloading files.

Upon successful start up the client searches a directory for available files and creates a dictionary using filenames as keys and file sizes as values. By default this directory is `/var/tmp/`. See **??** for more. Also, the client receives the user list from the server, and may commence with performing a search. It is important that a `download <filename>` request be preceded by an initial `\search <filename>` request as a download cannot commence without any search results.

Incoming server messages are parsed for the following commands, after which the appropriate action is taken.

- `__search <file name>` Perform a search on the local client.
- `++search <file names>` Receive search results back from a single client.
- `++download <key> <file> <address>` Perform an upload.
- `++pause` Pause the upload.
- `++resume` Resume the upload.

There is an additional `**download <key> <filename> <filesize>` command which is sent from one client to another to initiate a download as described below.

3.2.2 Searching

The client contains a `Searcher` object, which is responsible for searching through a clients local file list, and returning results to the server. The `difflib` library in Python was used to perform string matching, which matches close strings calculated from deltas between a query and a list of filenames. The advantage of this is that intensive processing is done client-side, rather than server-side, and is of particular concern for large file lists.

3.3 Encryption

The client performs encryption and decryption of keys sent and received as a result of file transfer requests. See **??** for more.

3.3.1 Downloading and Uploading

The client implementation makes use of several threads working concurrently to facilitate uploading a file, downloading a file, listening for server commands, updating the GUI, and updating the download and upload progress bars separately from other GUI elements. Downloading a file is performed by the `Downloader` object, which runs in its own thread. One such thread is started when the client is launched, and it takes care of all subsequent download requests. The `Downloader` thread listens for incoming connections from another client, and the `**download <key> <filename> <filesize>` command signals that a file download must be started. An `Uploader` object thread is created each time a client receives the `++download <key> <file> <address>` command from the server, signaling that a file must be uploaded to another client. At this point the `Uploader` thread establishes a connection with said client and sends the `**download <key> <filename> <filesize>` command. Once an upload has completed, the `Uploader` thread terminates its connection with the receiving client.

Clients can only perform one upload at a time and one download at a time (although these may occur together). Whenever a download or an upload is initiated, flags are set which prevent clients from uploading or downloading when such activities are already underway. When the download or upload finishes, the flags are set once more, to indicate a new download or upload activity may begin.

As mentioned, each progress bar is updated by its own thread. This is done by having the progress of an upload and download stored as separate global variables. Each thread updates its respective progress bar with the value from its respective global variable, at which point it sleeps to avoid updating the progress bar too often. The variable indicating upload progress is increment by the `Uploader` thread whilst the variable indicating download progress is incremented by the `Downloader` thread. These global variables are python lists which are made thread-safe via the Global Interpreter Lock (GIL).

In the `Uploader` thread, a requested file is opened and read as a binary file. The file is read in increments of 1024 bytes which are then sent to the receiving

client. This continues until the file is sent in its entirety. In the `Downloader` thread, a new file is created with the requested file's name and written to as a binary file in increments of 1024 bytes until all data have been received.

Pause/resume functionality was implemented by checking a variable in the `Uploader` thread. Clients who are downloading may issue a pause request to the uploading client, which sets the variable in the `Uploader` thread.

4 Complications

If the requested file already exists within the client's file directory, it will be overwritten.

While downloading and uploading takes place at acceptable speeds, once the downloading client issues a pause command, the uploading client slows down significantly. That is, if the uploading client is also performing a download, the download is slowed down significantly. This is believed to be caused by the while loop which allows for repeatedly reading and sending data. While the upload is on pause, the while loop does no real work during each iteration, and consumes CPU cycles. An attempted solution was to assign a low priority to the `Uploader` thread, but this approach was found to be ineffective, and abandoned.

Temporary connection failures do not impact the interaction between clients and server. Once the connection resumes, client-server interaction continues. For example, removing and reinserting the workstation's network cable does not cause clients to lose their connection to the server. Chat messages and search requests issued during the short period of disconnection are sent when the connection resumes and are still transmitted successfully.

5 Conclusion

The implementation affords an effective means by which to transfer files, with a distributed search and pause/resume functionality. The exercise was valuable in illustrating the concepts of peer-to-peer networks, and the special considerations that need to be made when keeping clients as anonymous as possible. This project has certainly been the most involved project in terms of programming— it required a lot of thread management, GUI related elements, network programming, and appropriate data structures. Thus, it has also been the most rewarding.