CS 313 Project 5

Group 31: P2P File Sharing

Keagan Selwyn Gill: 22804897@sun.ac.za

13/05/2024

Introduction/Overview

For this project, the requirement was to implement a peer-to-peer (P2P) file sharing program, that offered anonymity of the clients and security for the sharing of files. The framework was to have multiple clients connect to a single server. The server's role was to manage interactions between the clients by means of message passing (both text and commands). The clients, once introduced by the server, should then be able to transfer files P2P, from one client to the other without server interference. The client should have at least the functionality to support one concurrent upload and download each. The client would search for files with a server command which will return the results from each client connected. This client could then choose which file to download and begin after some security measures.

Unimplemented features

From the set of required features in the specification, all of them have been implemented.

Additional features

- Users can broadcast messages to each other, and they can select a subset of the connected clients to direct message (called whispering).

Description of files

Source Files:

ServerWindow class

- This class provides a GUI for displaying user activity information. It is responsible for establishing new client connections and forking a new client handler process for each new connection. The code generated by the Netbeans IDE which handles the GUI is also present in this class.

ClientWindow class

- This class provides the interface that the user interacts with. It is responsible for connecting to the server and maintaining all input and output from the server and other clients. It listens for input from other clients by forking a new ClientListenerThread process which handles the input from the other clients. Upon running this class, it prompts the user for their IPv4 address and the server's IPv4 address, then it initialises a new socket for the client's connection to the server, initialises the input and output

streams of the socket, then it prompts the user for a unique username and checks whether it is unique and handles it accordingly, then initialises some arrayLists and variables, runs the GUI code and finally calls a method to listen for messages. This class also contains the implementation for AES encryption for sending an encrypted key to securely transfer data. The code generated by the Netbeans IDE which handles the GUI is also present in this class. This class also contains the code, that had to be manually implemented, which handles the operations that occur upon clicking the buttons on the GUI.

ClientHandler class

- This class handles setting up a new client by getting a unique username from the user, reading in the client's port number, and adding this instance of ClientHandler into the clientHandler arrayList. It then listens for and handles incoming input from the associated user, whether it's a broadcast message to the other clients, a command, or a whisper message.

ClientListenerThread class

- This class runs on a new thread that listens for and handles input from the server and clients. It handles file uploads and downloads, file searches, and decryption of the message-key. It implements AES decryption for decrypting an encrypted message-key that gets sent along with user and file data each time a file is transferred.

Message class

- This class provides a custom object to use for sending messages between the server and clients as well as among clients by specifying the payload, who it is sent to, and who it sent from.

DownloadData class

- This class runs on a new thread that creates a peer-to-peer connection for receiving file data and handles the receiving of file data in chunks.

UploadData class

- This class runs on a new thread that connects to a peer-to-peer connection for sending file data and handles the sending of file data in chunks.

Program description

The server runs first and is responsible for establishing new client connections and forking a new client handler process for each new connection. The ClientWindow class is run when you want to add a new client instance. The ClientHandler initially sets up a new client with a unique username and port. The ClientWindow provides the graphical interface that the user interacts with. When the user enters input into text area and sends it, the associating ClientHandler class listens for it, receives it, and handles it. It does so by checking the prefix tag of the input to check whether it is a broadcast message to other clients, a command, or a direct message to specific clients, it then directs the message to the desired clients. The ClientListenerThread for each client listens for and receives this input and handles it accordingly based on the prefix

tag/command tag. It handles file uploads and downloads, file searches, and decryption of the message-key. The ClientWindow class contains the implementation for AES encryption for sending an encrypted key to securely transfer data. When a search request is sent from ClientWindow, it instantiates two new array lists, one for the users that have the searched file and one for the list of file name matches. It then passes this search request onto the ClientHandler where it will recognize that it is a search command and broadcast it to the other clients, where the corresponding ClientListenerThread's receive it, detects that it's a search command, runs the method to check for exact and substring matches, appends the matches to a results-command and then sends that back to the client that sent a search request where the found files will be displayed, if any. Then, if there are matches, the user can select the file that they want to send a download request for by typing in '/download ' with the corresponding index of the file and sending it by pressing enter. The sendMessage method in ClientWindow will detect that this is a download request message and will handle it by reading in the index chosen by the user, generate a random message-key and AES key, and then encrypt the message-key. It will then send off the message with the necessary user data, file data and encrypted message-key. The associated ClientHandler will then receive this message, detect that it is a directed message (by reading that it starts with the '@' character), so then the directMessage method will be called and will send the download request out to the desired client that has the file. The ClientListenerThread of the client who is being sent a download request will receive this download request, unpack the different segments of the message, display an option pane to ask whether that client accepts the download request or not. If yes is chosen, then a key-command prefixed with '/key' gets sent back to the client who requested a file download along with the encrypted message-key. The client who requested a file download then receives this message, decrypts the encrypted message-key and compares it against the original message-key. If they match, then a new download thread is started by calling the DownloadData class and then an upload thread is started on the client who received the download request by calling the UploadData class (on their end). The DownloadData class makes the client who is receiving the file a temporary server, the UploadData class then connects to this temporary server and uploads the file data in chunks using TCP, and the associated DownloadData class receives the file data in chunks.

Experiments

Experiment 1: Testing Concurrent Upload and Download Functionality

Question/Hypothesis: Can my peer-to-peer file sharing program effectively handle concurrent uploading and downloading of files, with each client restricted to sending one file at a time and receiving one file at a time, without significant performance degradation?

- Dependent Variable: Performance of file uploading and downloading.
- Independent Variable: Number of concurrent file transfers.

Experimentation:

1. Set up multiple instances of my peer-to-peer file sharing program on separate devices.
2. A 150MB file is used for receiving from another client and a different 150MB file is used for sending to another client.

3. Initiate concurrent file transfers, ensuring that each client sends one file and receives one file simultaneously.
4. Monitor the speed and efficiency of both the upload and download processes.
5. Measure the time taken for each file to complete the upload and download tasks.
6. Repeat the process three times.

Results/Conclusion:
The program effectively manages concurrent file transfers without significant degradation in performance, it can be concluded that our peer-to-peer file sharing program is capable of handling simultaneous uploads and downloads efficiently, even with the constraint of one file sent and received per client at a time.

Test	Client 1 (Upload to Client 2)	Client 2 (Download from Client 1)	Client 3 (Upload to Client 1)	Client 1 (Download from Client 3)	Client 2 (Upload to Client 3)	Client 3 (Download from Client 2)
1	15 sec	16 sec	18 sec	17 sec	20 sec	19 sec
2	17 sec	21 sec	16 sec	20 sec	19 sec	18 sec
3	16 sec	20 sec	20 sec	21 sec	18 sec	17 sec

Table 1: Representing experiment 1's tests of measuring transfer times of concurrent uploads and downloads for each client

Experiment 2: Testing File Search Functionality

Question/Hypothesis: Does the peer-to-peer file sharing program accurately display all corresponding search results from all clients containing files matching the search string?

- Dependent Variable: Accuracy of search results.
- Independent Variable: Search string used for file search.

Experimentation:
1. Set up multiple clients with the peer-to-peer file sharing program installed, each containing files.
2. Initiate a search query on one client using a specific search string (e.g., "music").
3. Verify that the search results display all files matching the search string from all connected clients.
4. Repeat the search process with different search strings (e.g., "documents", "videos").
5. Record the number of search results and compare them to the actual number of files matching the search criteria.

Results/Conclusion:
The search results accurately display all files matching the search string from all connected clients, regardless of the search term used, it can be concluded that the file search functionality of the peer-to-peer file sharing program is robust and effective.

Experiment 3: Testing File Integrity During Transfer

Question/Hypothesis: Does the peer-to-peer file sharing program ensure the integrity of files during transfer, without corruption or loss of data?

- Dependent Variable: File integrity.
- Independent Variable: File size and transfer method.

Experimentation:

1. Select a large file for transfer between two or more clients.
2. Monitor the file transfer process closely for any signs of corruption or loss of data.
3. Verify the integrity of the transferred file(s) by comparing them with the original file using checksums or file verification tools.
4. Repeat the transfer process with different files of varying sizes.
5. Analyze the results to determine if any files experienced corruption or data loss during transfer.

Results/Conclusion:
All transferred files maintain their integrity without any signs of corruption or data loss, it can be concluded that the peer-to-peer file sharing program effectively ensures file integrity during transfer.

Issues encountered

I ran into a number of errors and issues when trying to implement encryption and decryption. I eventually solved this however.

Design

A notable design choice was the set of algorithms put in place for securely sending and receiving file data:

Initially a client searches for a file they want to download by sending a search request, i.e., the '/search ' command with the search string. The search request is sent from ClientWindow, it instantiates two new array lists, one for the users that have the searched file and one for the list of file name matches. It then passes this search request onto the ClientHandler where it will recognize that it is a search command and broadcast it to the other clients, where the corresponding ClientListenerThread's receive it, detects that it's a search command, runs the method to check for exact and substring matches, appends the matches to a results-command and then sends that back to the client that sent a search request where the found files will be displayed, if any. Then, if there are matches, the user can select the file that they want to send a download request for by typing in '/download ' with the corresponding index of the file and sending it by pressing enter. The sendMessage method in ClientWindow will detect that this is a download request message and will handle it by reading in the index chosen by the user, generate a random message-key and AES key, and then encrypt the message-key. It will then send off the message with the necessary user data, file data and encrypted message-key. The associated ClientHandler will then receive this message, detect that it is a directed message (by reading that it starts with the '@' character), so then the directMessage method will be called and will send the download request out to the desired client that has the file. The ClientListenerThread of the client who is being sent a download request will receive this download request, unpack the different segments of the message, display an option pane to ask whether that client accepts the download request or not. If yes is chosen, then a key-command prefixed with '/key' gets sent back to the client who requested a file download along with the encrypted message-key. The client who requested a file download then receives this message, decrypts the encrypted message-key and compares it against the original message-key. If they match, then a new download thread is started by calling the DownloadData class

and then an upload thread is started on the client who received the download request by calling the UploadData class (on their end). The DownloadData class makes the client who is receiving the file a temporary server, the UploadData class then connects to this temporary server and uploads the file data in chunks using TCP, and the associated DownloadData class receives the file data in chunks.

Compilation and execution

Since this project was built using the Apache Netbeans 14 IDE, it cannot be compiled and run from the command-line/terminal. The project must be first opened and then compiled and ran from within the Apache Netbeans IDE. To do so, follow these steps:

1. You must have the Apache Netbeans 14 IDE installed on your computer.
2. Open the IDE.
3. Move your cursor to the top left of the application and click on “File” and then “Open Project” (or use the keyboard shortcut Ctrl+Shift+O).
4. Navigate to the directory of your desired project that you want to open.
5. Select the project folder that contains the folders generated by Netbeans upon creating a project, namely “nbproject” and “src”. In this case the project folder is called “P2P-File-Share” and it should have a little coffee-cup-on-a-saucer icon to the left of it indicating that it is a Java project. Once selected click on “Open Project” at the bottom right of the window.
6. To run the Server, first click on the ServerWindow.java file to open it, **click on any line of code in the file (so that Netbeans knows that this file is the file you want to run), then either left click to open a drop-down list and click on “Run File” or use the keyboard shortcut Shift+F6**. Now the built-in terminal will be displayed and the GUI window for the Server will also be displayed.
7. To run the Client, first click on the ClientWindow.java file to open it and then repeat the process in step 6 above that is **bold**. Now the built-in terminal will be displayed and the GUI window for the Client will also be displayed.

Libraries used

- java.io.IOException
- java.net.ServerSocket
- java.net.Socket
- java.awt.event.ActionEvent
- java.awt.event.ActionListener
- java.io.File
- java.io.ObjectInputStream
- java.io.ObjectOutputStream
- java.lang.reflect.InvocationTargetException
- java.net.InetAddress
- java.net.UnknownHostException
- java.nio.charset.StandardCharsets
- java.util.ArrayList
- java.util.Base64

- java.util.logging.Level
- java.util.logging.Logger
- javax.crypto.Cipher
- javax.crypto.KeyGenerator
- javax.crypto.SecretKey
- javax.swing.JFileChooser
- javax.swing.JOptionPane
- javax.swing.filechooser.FileSystemView
- javax.swing.*
- java.sql.Timestamp
- java.nio.charset.StandardCharsets
- javax.crypto.Cipher
- javax.crypto.SecretKey
- java.io.Serializable
- java.io.FileOutputStream
- javax.swing.JProgressBar