JT Reagor, Thatcher Crist, Connor Talbot

CS 371 Project

Remote Folder Application

Abstract

This report will discuss the use of sockets and the TCP/IP protocol to establish a connection to a secured, private folder held on a server. Our application works well across both wi-fi and ethernet. It can handle multiple clients at the same time, but it runs into problems with users accessing the same file at the same time.

Introduction

Remote folders are used incredibly frequently in many aspects of modern-day life. Through internet protocols, we can relatively abstractly implement connections across vast distances. This allows us to access shared information from any location. We are no longer bound by where we are for what information we can use and edit. The goal of this project was to create a very basic remote folder that a client can upload to, download from, and access information about. The folder should also be able to manage multiple connections from incoming clients.

The client-side user establishes a connection with the command “CONNECT <server ip> <server port>”. The server then enforces username and password authentication. It is not incredibly secure because the passwords are stored in plaintext in the code and are transmitted unencrypted. However, with ease, we could implement or use some form of encryption. Upon successful login, the server creates a thread to handle each client. Then, by a command line in the client, the user can perform a certain set of commands to manipulate the folder. These commands are as follows:

* LOGOUT
  + Disconnects client and kills thread in server
* ECHO
  + Sends a response message to see if connection is truly established
* DIR
  + Returns information about the files in the shared folder: file name, number of downloads, upload date
* DELETE <filename>
  + Deletes the file from the shared folder
* UPLOAD <filename>
  + Uploads a file from the client’s directory to the shared folder
* DOWNLOAD <filename>
  + Downloads a file from the shared folder to the client’s directory

Implementation

The server and client interactions are intertwined relatively tightly, but I will attempt to describe the processes and implementations of both separately in the following section.

We used Python to implement our remote folder due to the ease of access to libraries and simplicity of the language. With a higher-level language, we did not have to worry about all of the painful steps that come with languages like C.

Client-side implementation

Upon startup, the client establishes a socket using IPv4 and the TCP connection-oriented protocol. Then, the client enters one large loop that establishes a terminal interface. This terminal takes commands and handles them according to their desired function. The first command is CONNECT. This takes 2 arguments: the server ip and the port. Using the sockets connect function, it establishes a connection between the server and client. It then expects a welcome message response to confirm the connection.

The next command is LOGOUT. The client sends a packet to the server saying that it is disconnecting, receives a goodbye message, and resets the socket. The client then re-enters the main loop with the reset socket.

The command ECHO simply sends a message to the server asking for a response. This response is then displayed in the terminal to prove that the connection works.

The command UPLOAD is where things start getting interesting. UPLOAD takes a file name as an argument. The client sends a message to the server declaring the length in bytes of the file to come and the name of the file. Then, after receiving an ack declaring the server is ready, the client enters a loop to send the data stored in the file in chunks of 1024 bytes. The client reads 1024 bytes at a time into a buffer and sends the buffer to the server. The client then waits for an ack saying the server is ready for more data and repeats until all of the data is sent.

The command DOWNLOAD is nearly an exact copy of the server’s response to the UPLOAD command. DOWNLOAD takes a file name as its argument. It then sends a request to the server that contains the file name. The client then waits for the server to send back the size of the file to be expected. The client then creates and opens a new file with the name of the file. It then enters a loop to receive the data. In this loop, it receives a chunk of data and puts it into a buffer. Then, it sends an ack saying that it is ready for more data. This continues until the size declared at the beginning has been reached. Then, the buffer is written to the open file.

The command DELETE sends a request to the server to delete the specified file. The client then waits for a response from the server as to whether the deletion failed or not.

Evaluation

Conclusion