Server and Client Report

The following report is a description and evaluation of a solution to building a file service application that allows the user to upload files to or download files from a server as well as list files that are stored on the server side. This solution consists of three Python scripts, server.py, client.py, and shared\_process.py.

The server.py script is executed through the Windows command line and takes in a port number as an argument. It appends the parent directory to the sys.path, imports the relevant functions from the shared\_process.py script (which will be expanded upon later) and then appends the server directory, where the script itself is stored.

Next it creates a TCP server socket and uses a try statement to declare a hostname and a port. The server is then bound to the port number given in the user argument. The hostname is defined as “0.0.0.0”, a special IP address that refers to all IP addresses, because then the program can receive messages at the port number on any interfaces or addresses that the user’s computer may have.

If this is successful, this is printed out and the server waits for client connections. However, if it isn’t, the except is called so that the error is printed and the program can be exited safely.

A while True loop is then utilised to handle requests because the program should run for as long as there is something to connect to. A try statement is used and the client’s socket and address are then connected to the server and the receive\_data() function from the shared\_process.py script is called. It {}

The request is then parsed so that the program can handle the arguments. If the request is “list”, (which lists the first level directory contents), the build in function os.listdir() is used to get the contents. This contents is then edited by “\n”.join(dir\_contents\_list) which concatenates each element of the list onto a new line so that the output is more readable and useful to the user. From the shared\_process.py script, send\_listing() is called and if everything has worked a line is printed saying so to keep the user informed on what is happening.

If the request is “get” (which downloads a file to the client), an if-else statement is used to check that the file exists, which allows the user to receive an informative error message if the file is not there, preventing confusion when the program crashes. When the file is there, send\_file() is called from the shared\_process.py script, and a line is printed to say the request has been successful.

Finally, if the request is “put” (which uploads a file), the handling is very similar, with the main difference being that the recv\_file() function is called from shared\_process() instead.

If there has been a larger error within the code, the except part of the initial try statement after the while True loop handles that and prints the error, so that the user has the information help them run the program successfully.

The ‘client.py’ script is also executed through the Windows command line and imports the relevant functions from shared\_process(). It creates a client socket and then uses a try statement to parse through the three arguments given in the command line (not including the script itself).

The first argument is the hostname, the next is the server’s port number and then the calling word that will enact one of the actions.

If the user requests to upload or download a file the program checks that the user has provided a file name to be uploaded or downloaded by checking that five arguments have been entered into the command line. If there isn’t a fifth argument, an IndexError is raised, the user is reminded that a file name must be provided to run those requests so that they can be successful next time. The IndexError is then excepted as part of the initial try statement, and the user is told that an incorrect number of arguments were provided.

To prevent the user repeatedly requesting an actions that don’t exist, and if statement is used so that if an invalid request is entered an Exception is raised, informing the user that their request is invalid and reminding them of the valid requests. Any other errors will also be excepted, printed out for the user’s understanding, and then the program will exit safely.

The server address is declared a tuple of the hostname and port and connected to the client socket and a line is printed to inform the user that the connection has been successful to keep them informed on the program’s progress.

A try statement begins the section of code that handles the requests and it is closed by an except Exception statement which will handle and print any errors that are not caught by previous error checking, allowing errors to be more easily understood by the user.

\*\*\*STILL TO FIX\*\*\*

If the choice is list, the program encodes the string and sends it, it the choice is “get” or “put” another if statement is used first to check the the file exists, and then either the recv\_file() or get\_file() function is called respectively.

After receiving and processing the server’s response it closes the connection.

Our ‘shared\_process.py’ script is used in the interest of clean code. Calling functions from a separate script which solely contains functions to be called by the main scripts makes the main scripts more readable as it isn’t clogged up with more code than is necessary.

The script contains four functions. The first is send\_file() which reads the file with the same name that is provided by the user. It makes use of a try statement and two except statements one which handle a broken connection called BrokenPipeError and another which handles any other errors and prints a line with the error and another explaining that the upload failed.

The next is ‘recv\_file’ which creates a file with the name provided by the user and writes to the data the socket has received. It also uses a try statement with the exceptions of a BrokenPipeError and any other that may occur.

Send\_listing It is encoded and sent to the client socket and