Networks and Operating Systems Essentials 2 Assessed Exercise 1 Report

Gregor Johnston	2773209J	LB02
Callum Cooper Lee	2758297C	LB08

After a confirmation with lecturer Awais, we were informed that we could work with each other in different lab groups if we both attend the same lab session for evaluation.

Introduction

The specification of our project was to create a server and a client that could transfer files to one another using TCP and UDP protocol communication. We implemented this program in python, with a 'server.py' and 'client.py' stored in their respective 'server' and 'client' directories. An 'assignment.py' was implemented containing methods to be used by both python files. This report documents our Design of the application-level protocol.

Design of the application-level protocol

Client.py:

- First, the client.py file was developed, as it is the base were filenames and commands would be sent from. Connection code was setup.
- The list of commands was defined as a list ["put","get","list","exit"]. WE went with this method of command so that when the user inputs a command, the code can check if command in commands and stop if command is invalid (which includes if command == null).
- For filename, it was okay if no filename was input since list and exit do not require filename parameter. Then, within a try block, a connection to the server is attempted, with a success message printing in a success, and a FAILURE REPORT if not.
- For command and filename to the server we made a new String variable "datasend" which is a formatted string of command and filename, separated by a "\n" character. If an error arises about the filename not being defined, then the try block will move to its except, where it handles the filename error by only sending the command to the server.
- WE used send() over sendall() here to send the data. We did this because send() uses the TCP protocol, which is better suited for low data sending, since all its sending are two small Strings which take up very little space. sendall() uses UDP protocol and runs a higher level function of sending until all data is sent, which is unnecessary for this example. That is why we chose send().
- The next part is choosing which command is to be run, implemented with a simple if elif else block.
- My solution to the issue of overwriting files is to simply get the current directory of where the current python file is stored, store it in a variable current_directory (so current_directory is now a list of Strings of the names of the files), and check that the filename is not in the list of files in current_directory by using <u>if filename in current_directory</u>. So when downloading files, it checks if the current directory already has that file and sends a FAILURE REPORT if it does.
- This way it also can check that when sending a file, the file being sent is a file that is in the current_directory, which solves the issue of incorrect filenames. So for the command "put", you would be checking the filename exists in current_directory, then go ahead and try to send it.
- Once a command has been executed, it will print out its report on what happened and the client socket will then be closed using cli_sock.close(), and it will then exit(0)

Server.py:

- Next, the server.py was created. Connection code was setup.

- We implemented a while loop, so when the server is started, it will stay open forever unless there is a fatal error or the server is forcibly closed by a command (“exit”).
- Then, in a try and finally block, server waits for a client to attempt to connect. Once this happens, a print method reports on the success of the connection by printing the IP address and the port number of the client.
- A variable is initialised in server.py as data_recieved, receiving the bytes sent in “datasend” from the client.py. It looks like data_recieved = cli_sock.recv(1024).decode()
  Data is received from client socket, uses the .recv() function with number corresponding to max number of bytes it will receive and .decode() converts bytes to String. Implementation was selected to correspond to client send(). The byte size is 1024 since it is very unlikely the filename will and command will exceed 1024 bytes of data.
- Also corresponding to the send, if the code runs into an error with the split not working because no “\n” was scanned, the try: block moves to its except, where only command is sent and received (for “exit” and “list” since no filename is necessary)
- Server.py uses the same implementation of checking a file is in directory that client.py does (Can’t send a file that doesn’t exist in current directory and can’t receive a file that already exists in current directory)
- Once the command has been selected and its corresponding function has been executed, the cli_sock is closed and the while restarts its loop awaiting a new connection.

<u>Assignment.py:</u>

- The send_file() function was written in assignment.py in order to be used by both the server and the client as they will both be sending files. It operates using the UDP protocol with a sendall() to ensure that all the data is sent. Once the entire file has been sent a send(“<END>”) was utilised in order to mark the end of the file.
- The recv _file() function was written in assignment.py in order to be used by both the server and the client as they will both be receiving files. The function recv() can only receive 1024 bytes at a time. It was placed in a while loop in order to ensure all data is received through the socket. The send(“<END>”) was utilised in order to mark the end of the file, so the received data is checked every loop until the “<END>” is detected and the transmission has therefore terminated.
- send_list() stores the top layer directory into a string separated by “\n” new lines and sends with a sendall(). This is received and interpreted by recv_list().