CITS3002 Report

Date Submitted: 25/05/2022
Date Due: 25/05/2022, 5pm
Lecturer: Dr Chris McDonald
Unit Code: CITS3002
Unit Name: Computer Networks

Student Name	Student Number
Thanh Nguyen	22914578
Anfernee Pontilan Alviar	22886082
Ethan Pui	22704879

Protocol Design

Encoding type

Integers will be encoded into 8 bytes using the big edian byte order, where the most significant bytes will be stored first, at the lowest storage address. Strings will be encoded using the utf-8 standard.

When communications are made between client and server, the client must make the initial connection, and the first type of communication must be represented as a 4 byte integer followed by any number of payloads. Once the entire datagram is sent by the client, the client will be in wait mode until an acknowledgement has been received from the server, then more datagrams can be sent, in the same order of a 4 byte integer followed by one or more payloads.

The 4 byte integer will prompt the server or the client on what kind of payload to expect, this will set the receiving connection into a state in which it can properly accept the incoming payload. Standard strings will be encoded in utf-8 bytes before sending and decoded on the other side.

Cost Request Protocol

When the initial connection is accepted by the server the client will send an integer represented by CMD_QUOTE_REQUEST padded to 4 bytes in big edian byte order.

Once the preamble is accepted by the server it will return an integer represented by CMD_QUOTE_REPLY followed by the cost also in an integer and all padded to 4 bytes and using the big edian byte order, refer to figure 1 diagram.

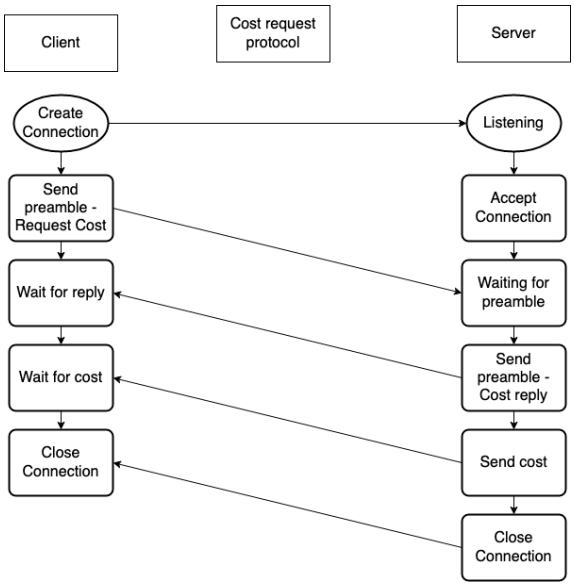

Figure 1

Send File Protocol

Once the appropriate file is located and buffered into memory, the sending connection will first start by

1. sending the preamble CMD_SEND_FILE if the file is a text file or CMD_BIN_FILE if the file is binary.

2. The size of the name of the file, followed by the name of the file, formatted in utf-8 encoding

3. The size of the actual file following our established 4-byte big edian standard.

4. Send the contents, if in text format it will be encoded in utf-8 otherwise the binary will be sent.

5. Wait for an acknowledgement from the server - CMD_ACK - to prompt the client the server is ready for the next file.

6. If more files need to be sent repeat 1-4

Execute Command Protocol

Once the server has all required files, the client will do the following.

1. Send the preamble CMD_EXECUTE

2. The command will be in string format so we first encode it in utf-8 bytes and send the size, followed by the command.

3. Wait for a return code, this will be received as an integer

4. If the return code does not equal zero, something went wrong and the server will send the error message.

   - The server will send the size of the message as an integer followed by the message, the message will have the utf-8 encoding

5. If the return code equals zero, the client will wait the file. Protocol for receive file below.

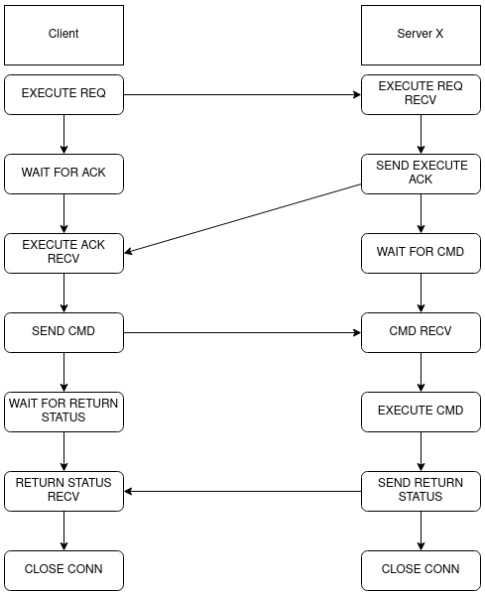

Figure 2

Receive File Protocol

The receiving socket will be in a state to receive incoming data.

1. Evaluate the preamble.

   - CMD_BIN_FILE

   The receiver will expect to write bytes to a file

   - CMD_SEND_FILE

   The receiver will expect to write ut8-8 chars to a file

   - CMD_RETURN_FILE

   The client will expect a binary file and write bytes to a file

2. The preamble will prompt the receiver on what kind of file to expect.

3. The size in bytes of the name of the file will be sent, then the name will be sent encoded in utf-8 bytes

4. The size in bytes of the contents, followed by the contents.

   - if the receiver is expecting text CMD_SEND_FILE it will simply decode and write to a text file.

   - otherwise, the receiver is expecting a binary file, in that case, the receiver writes the byte directly to the file.

5. If the receiver is in CMD_RETURN_FILE mode it will simply close the connection, as this mode is only ever each at the end of an action set. Otherwise, a CMD_ACK will be sent and repeat steps 1-5

Walkthrough of server and client interactions

1. Servers are created by running (python3 rakeserver.py 6328). Opening an port based on the command line third argument and listens for connections on that port number and IP address.

2. Client is created by running (python3 rake-p.py Rakefile). Client creates a connection to a given host on a given port given to the function *create_socket* as a parameter. Client then connects the socket and returns the connection object in the *create_socket* function for quote while listening for servers.

3. When the connection is established with a server the client will initiate the connection in the handle_conn function by sending a preamble represented in 4 bytes and edian byte order CMD_QUOTE_REQUEST.

4. The server that is waiting for this preamble receives CMD_QUOTE_REQUEST and sends back a preamble CMD_QUOTE_REPLY in the *send_quote* function. If the preamble is not correctly converted on the client side the server will not be able to recognize the preamble and thus do nothing.

5. The client waiting for a reply receives the preamble CMD_QUOTE_REPLY in the *handle_conn* function and waits for a second reply of the cost in the edian byte order. The server sends the cost and closes the connection, the client receives the cost and closes the connection.

6. Steps 3 to 5 are repeated until the client has found server with lowest cost.

7. Client creates and connects sockets for executing commands on the server at the lowest cost. The client determines if it requires files by reading the Rakefile. If it does require files, the client sends the required files to the server by calling the function *send_txt_file* and sending a preamble CMD_SEND_FILE in big edian byte order.

8. Client will return code status when running *handle_conn* function detailing if there was an error or success when executing commands. On success, the server will send a file and the client will receive the output file from the command. If it was an fatal or warning error, the client will output an error message detailing the condition and exit the program immediately. The execution of the commands processes is shown in the diagram in figure 3.

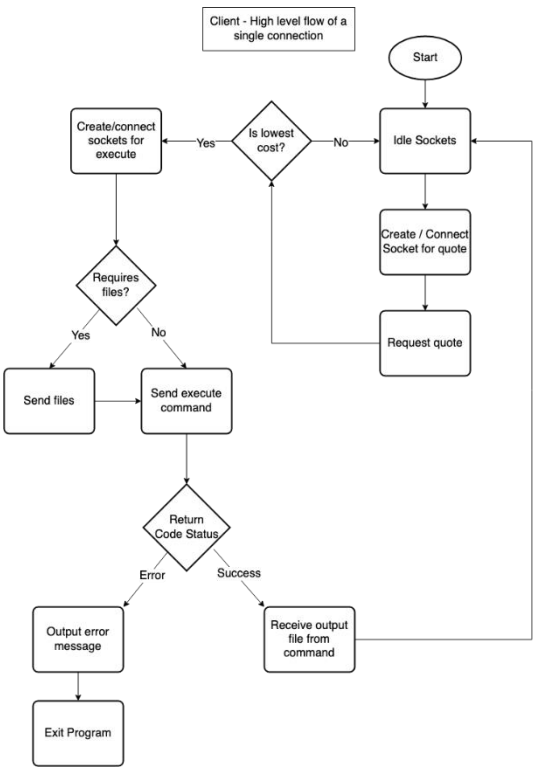

Figure 3

Performance

When compiling sufficiently large programs on a local machine, the compilation process completes in sequential order. Therefore, if an object file does not have any dependencies, it still must wait for its turn to compile. When the remote compilation is possible, we can decompose the requirements into "action sets" and have a network of computers run in parallel the compilation process for individual object files.

For example, assuming a compilation where the object file does not have many dependencies such as the program.c example then we can send just the files needed to create the object file and have the local machine perform the final compilation.