# Chat Design Work - Program #2

**Derek J. Russell**
**April 14th, 2022**

## Part 1:

### Packet Header Structure

Flag = 1 Sent from cclient to server - Client initial packet to the server

| | | | |
|---|---|---|---|
| PDU Len<br>2 Bytes<br>Network Order | Flag = 0x01<br>1 Byte | Handle Len<br>1 Byte<br>Network Order | Handle<br>X Byte(s)<br>Network Order |

Flag = 2 Sent from server back to cclient - confirming a good handle

| | |
|---|---|
| PDU Len<br>2 Bytes<br>Network Order | Flag = 0x02<br>1 Byte |

Flag = 3 Sent from server back to cclient - Error on initial packet (handle already exists)

| | |
|---|---|
| PDU Len<br>2 Bytes<br>Network Order | Flag = 0x03<br>1 Byte |

Flag = 4 / %B Sent from cclient to all other clients via the server - Broadcast packet

| | | | | |
|---|---|---|---|---|
| PDU Len<br>2 Bytes<br>Network<br>Order | Flag =<br>0x04<br>1 Byte | Length of<br>the<br>sending<br>clients<br>handle<br>1 Byte | Handle<br>name of<br>the<br>sending<br>client<br>1 Byte | Payload: Message<br>200 Bytes |

Flag = 5 Sent from cclient to another cclient via the server. This is a message (%M) packet

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| PDU Len<br>2 Byte<br>Network<br>Order | Flag = 0x05<br>1 Byte | Length of<br>the sending<br>clients<br>handle<br>1 Byte | Handle<br>name of<br>the sending<br>client<br>1 Byte | Number of<br>destination<br>1 Byte | Length of<br>handle of<br>the<br>destination<br>client<br>1 Byte | Handle<br>name of<br>the<br>destination<br>client<br>X Byte(s) | Payload:<br>Message<br>200 Bytes |

Flag = 6 (no longer used)

Flag = 7 From server to cclient - Error packet, if a destination handle in a message packet (flag = 5) does not exist.

| | | | |
|---|---|---|---|
| PDU Len<br>2 Byte<br>Network Order | Flag = 0x07<br>1 Byte | Handle Length<br>1 Byte | Handle name of the<br>destination client<br>1 Byte |

Flag = 8 / %E Client to server when the client is exiting (client program cannot terminate until it receives a flag = 9 from the server)

| | |
|---|---|
| PDU Len<br>2 Byte<br>Network Order | Flag = 0x08<br>1 Byte |

Flag = 9 Server to client, ACKing the clients exit (flag = 8) packet

| | |
|---|---|
| PDU Len<br>2 Byte<br>Network Order | Flag = 0x09<br>1 Byte |

Flag = 10 / %L Client to server, client requesting the list of handles. (So %L command)

| | |
|---|---|
| PDU Len<br>2 Byte<br>Network Order | Flag = 0x0A<br>1 Byte |

Flag = 11 Server to client, responding to flag = 10, giving the client the number of handles stored on the server.

| | | |
|---|---|---|
| PDU Len<br>2 Byte<br>Network Order | Flag = 0x0B<br>1 Byte | Number of handles at the server<br>4 Bytes<br>Network Order |

Flag = 12 Server to client, immediately follows the flag = 11 packet. Each flag =12 packets contain one handles currently registered at the server. There will be one Flag =12 packet per handle.

| | | | |
|---|---|---|---|
| PDU Len<br>2 Byte<br>Network Order | Flag = 0x0C<br>1 Byte | Handle Length<br>1 Byte | Handle name of the destination client<br>1 Byte |

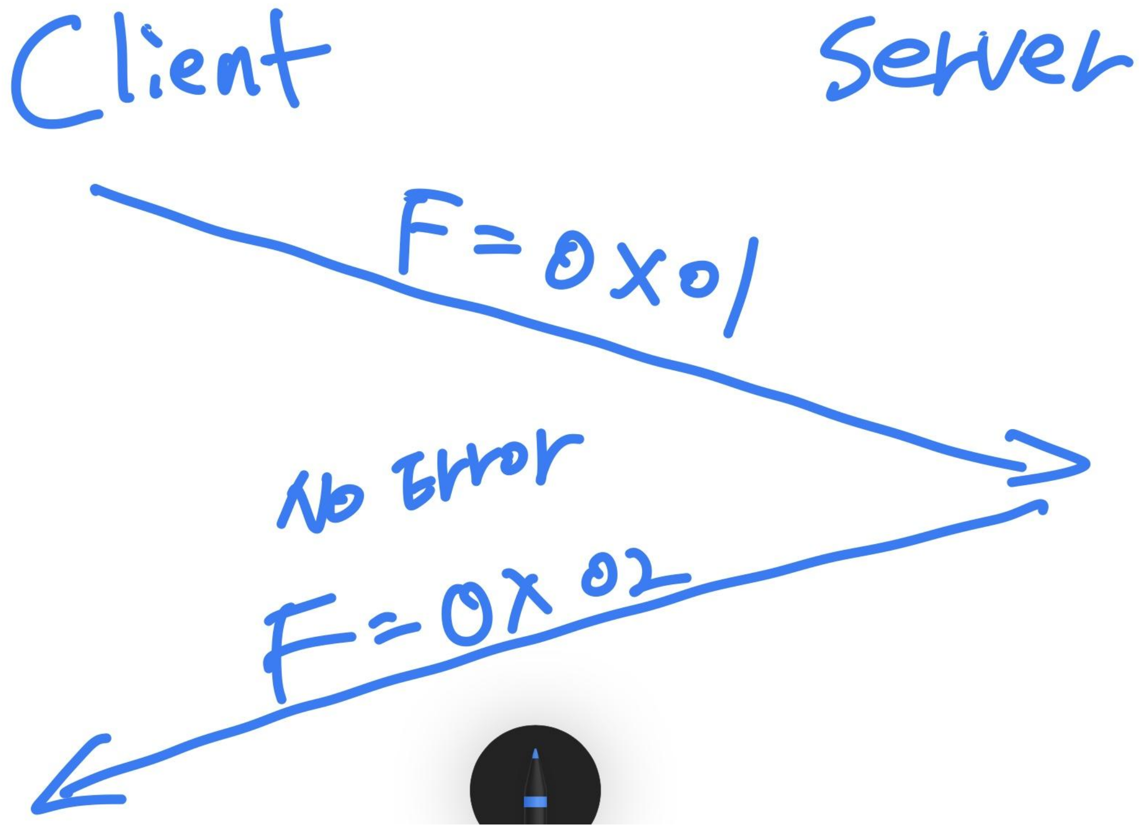
Flag = 13 Server to client. This packet tells the client the %L is finished.

| | |
|---|---|
| PDU Len<br>2 Byte<br>Network Order | Flag = 0x0D<br>1 Byte |

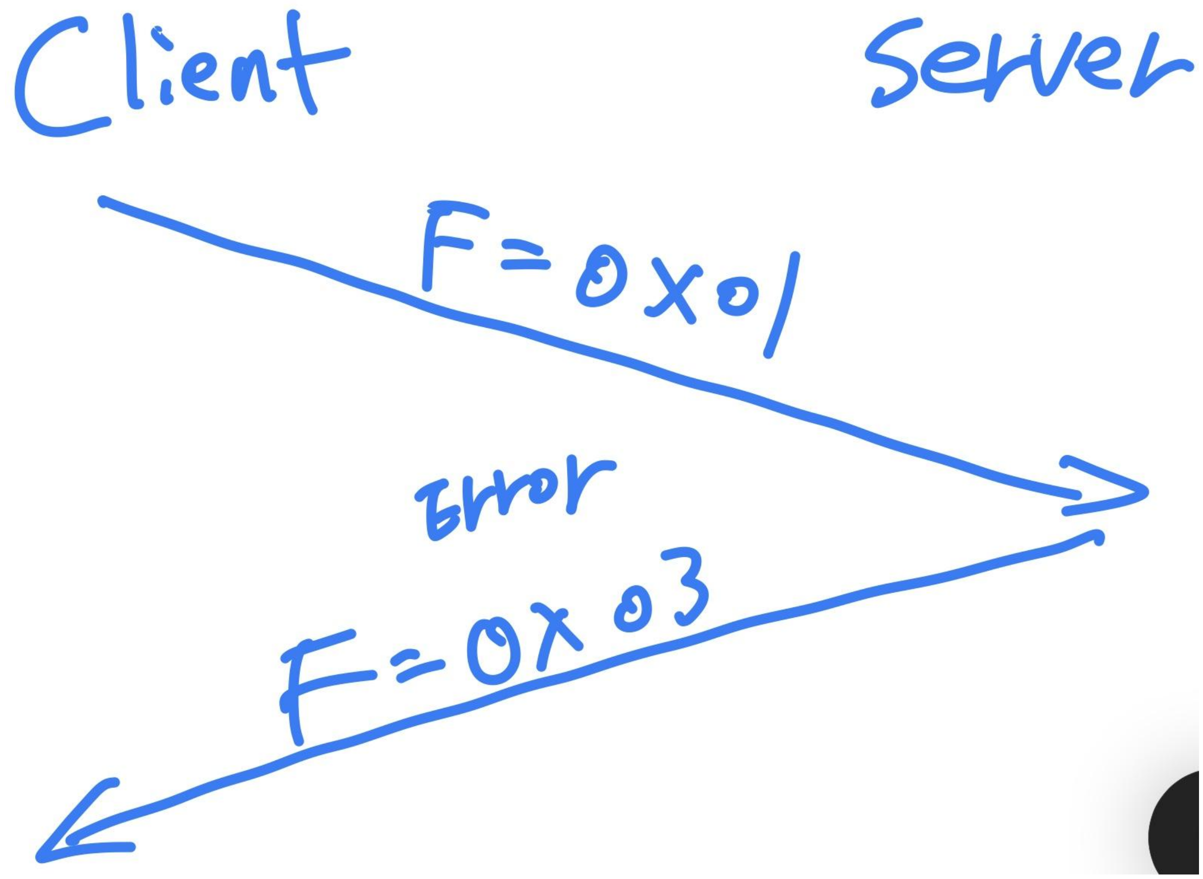
## Pack Flow Diagram

a. Connection setup

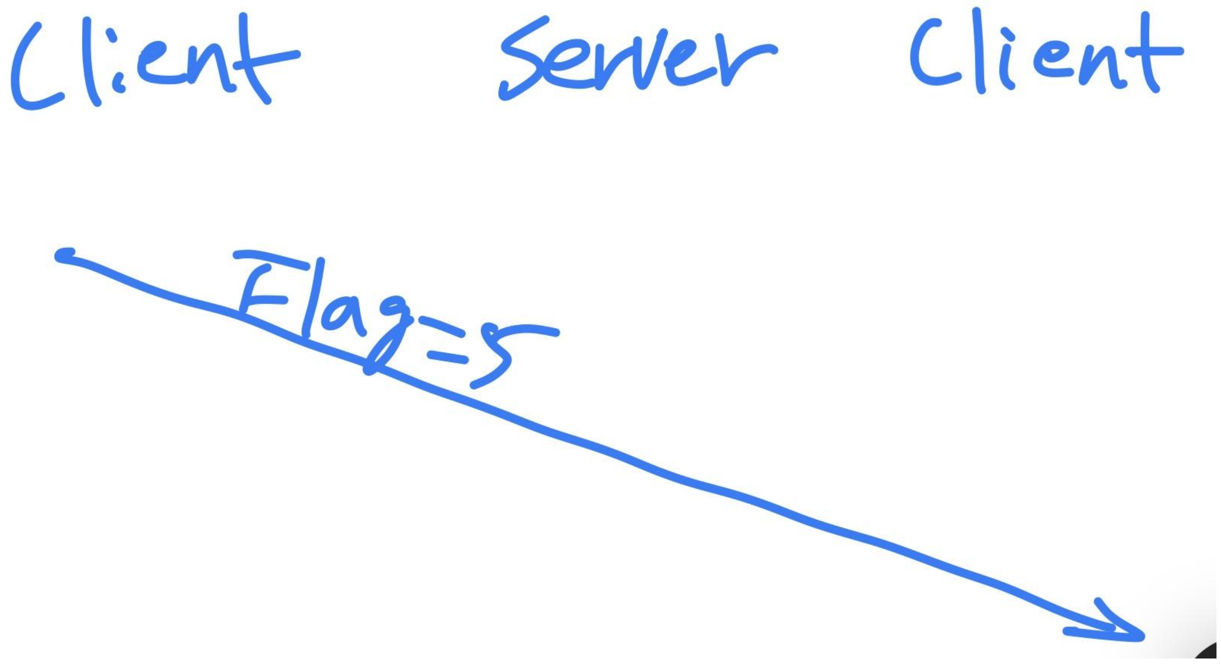
i. **Description:** The client connects to the server with no error.
**Processing on client:** The client sends a packet with the flag set to 0x01.
**Processing on server:** The server sends a packet with the flag set to 0x02



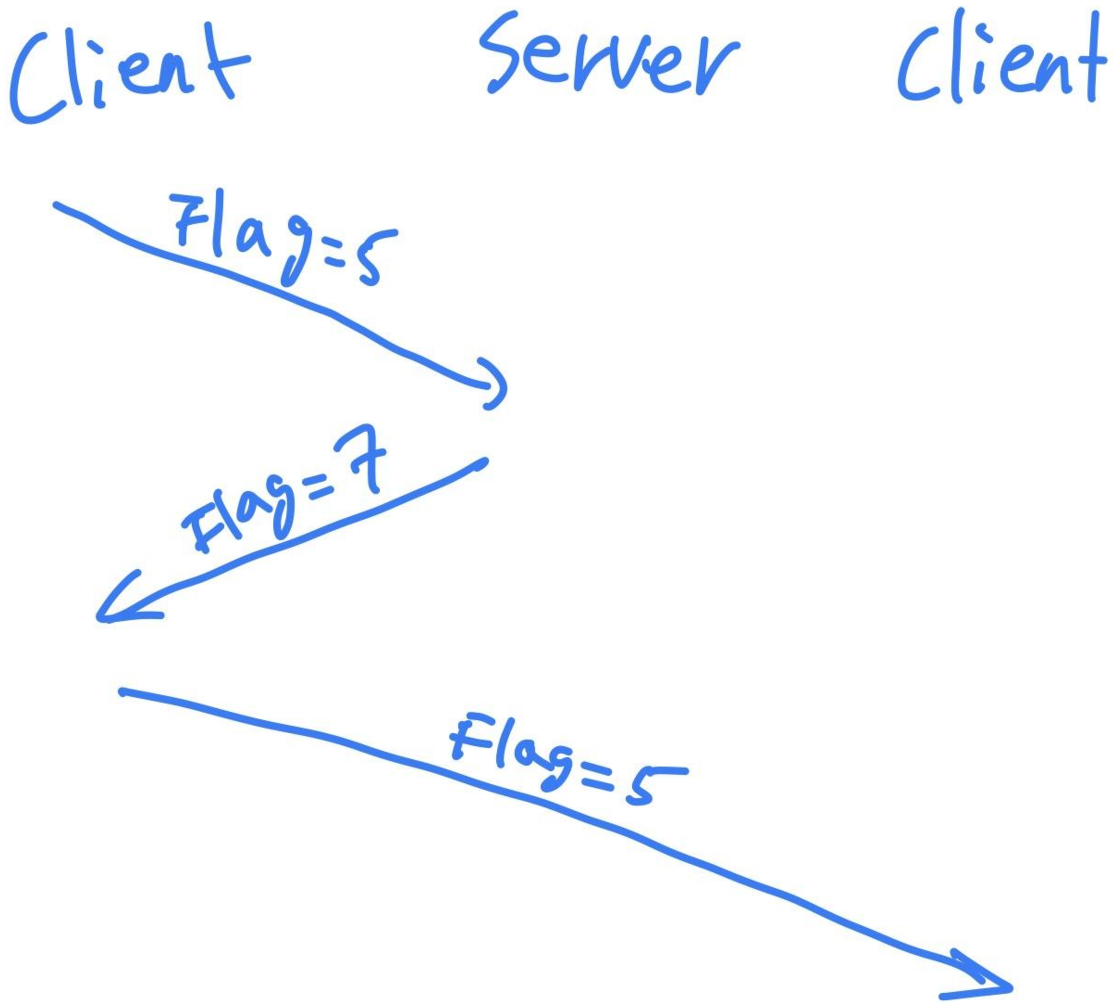
ii. **Description:** The client connects to the server with an invalid handle
**Processing on client:** The client sends a packet with the flag set to 0x01.
**Processing on server:** The server sends a packet with the flag set to 0x3.
**Processing on client:** The client prints an error message and terminates.



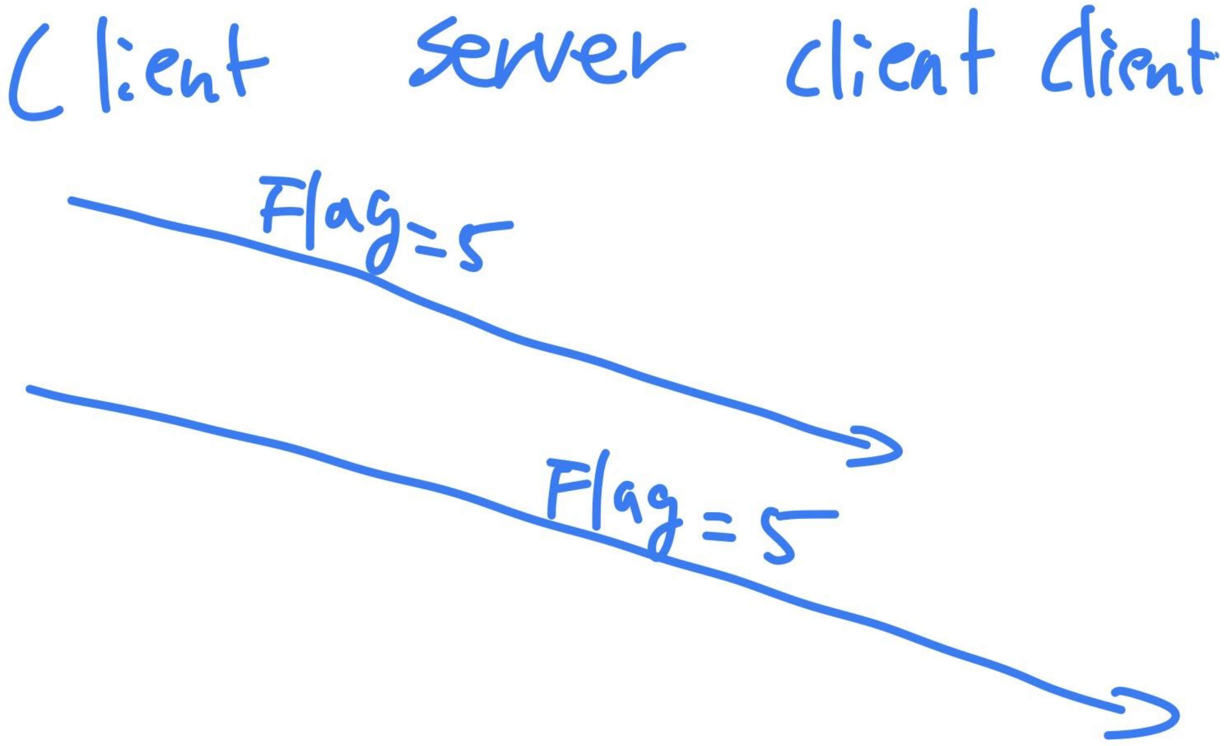
b. Sending a message (%M) to a single client:
    i. **Description**: The sending client successfully sends a message to the destination client.
    **Processing on sending client:** The client sends the packet containing the message with the flag set to 0x5.
    **Processing on server:** The server forwards the packet to the destination client.
    **Processing on destination client:** The client writes the received message to STDOUT


Client
Server
Client
Flag=5

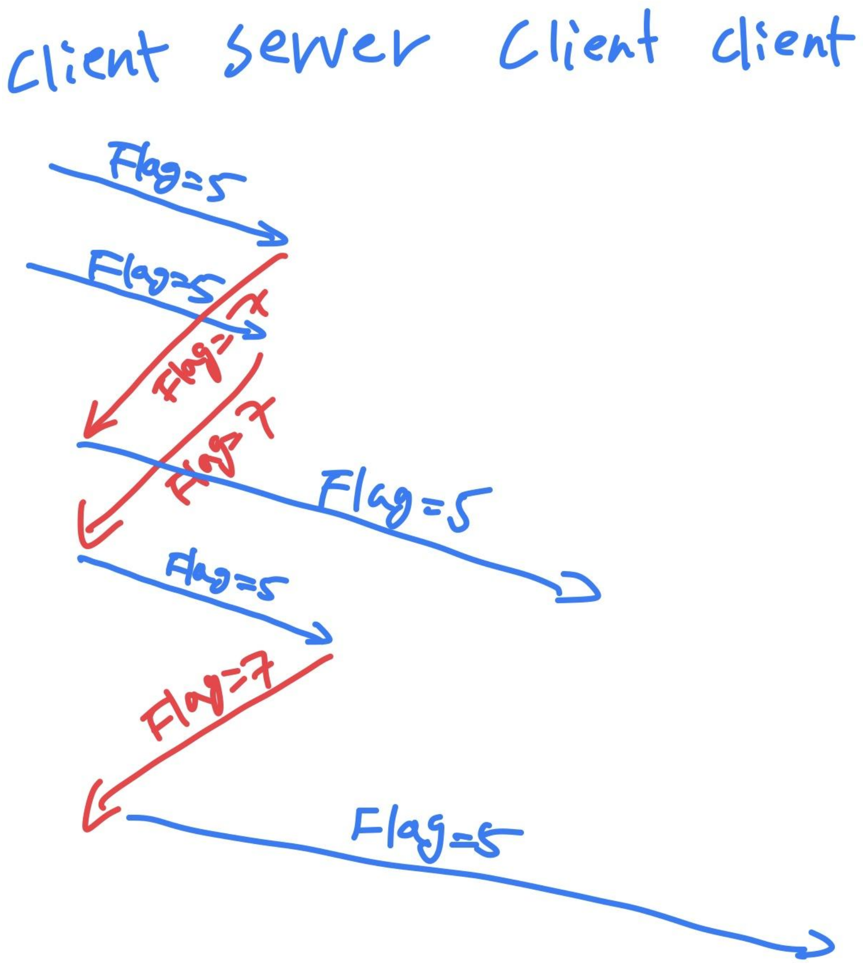
ii. **Description:** The sending client unsuccessfully sends a message to an invalid client.
**Processing on sending client:** The client sends the packet containing the message and destination client handle with the flag set to 0x5.
**Processing on server:** The server sends a packet to the sending client with the flag set to 0x7.
**Processing on sending client:** The client prints out an error message, specifying the invalid client handle.



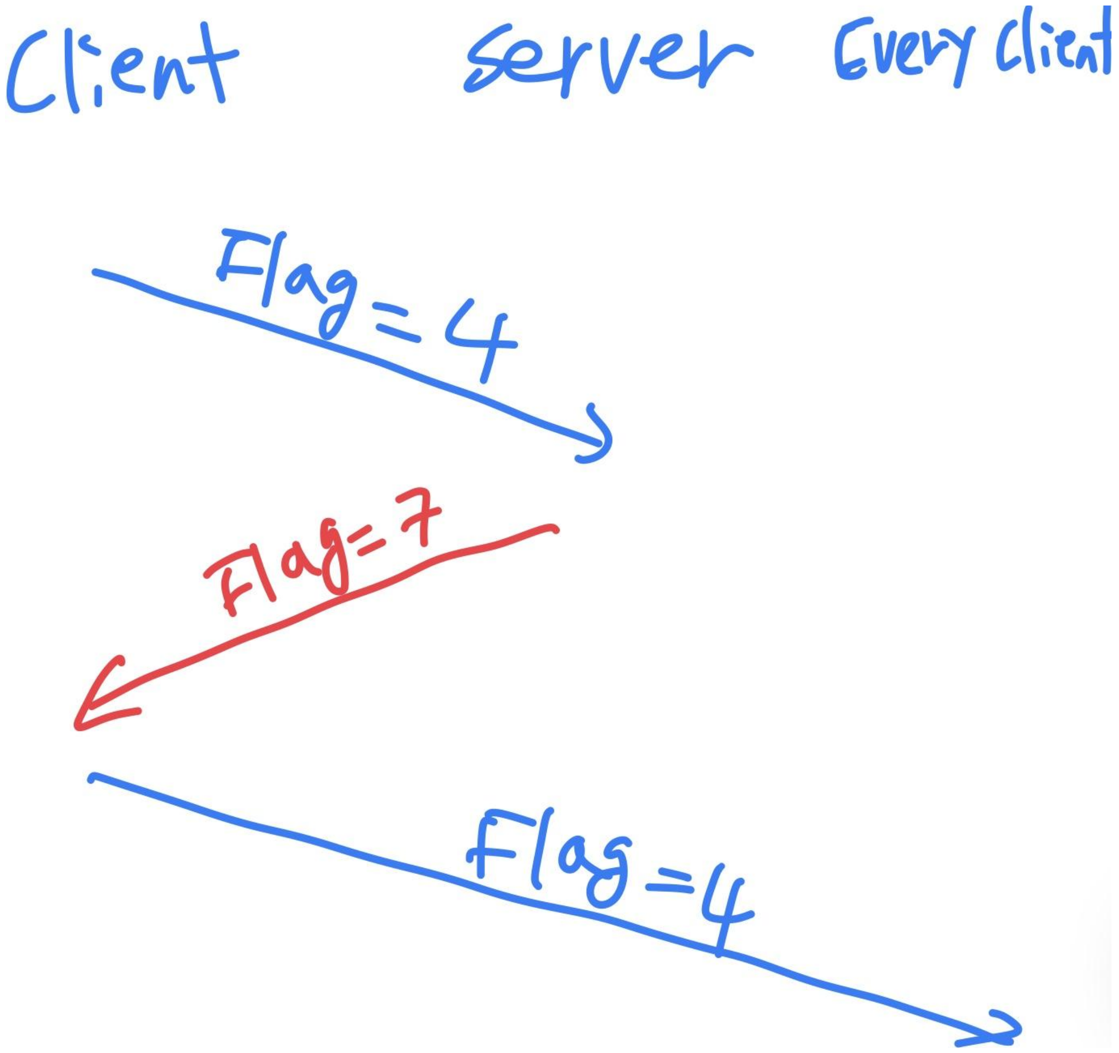
c. Sending a message (%M) to multiple clients:
   i. **Description:** The sending client successfully sends a message to the destination clients.
   **Processing on sending client:** The client sends a packet containing the message and destination client handles with the flag set to 0x5.
   **Processing on server:** The server forwards the packet to the destination clients specified in the packet.
   **Process on destination client:** The destination client writes the received message to STDOUT



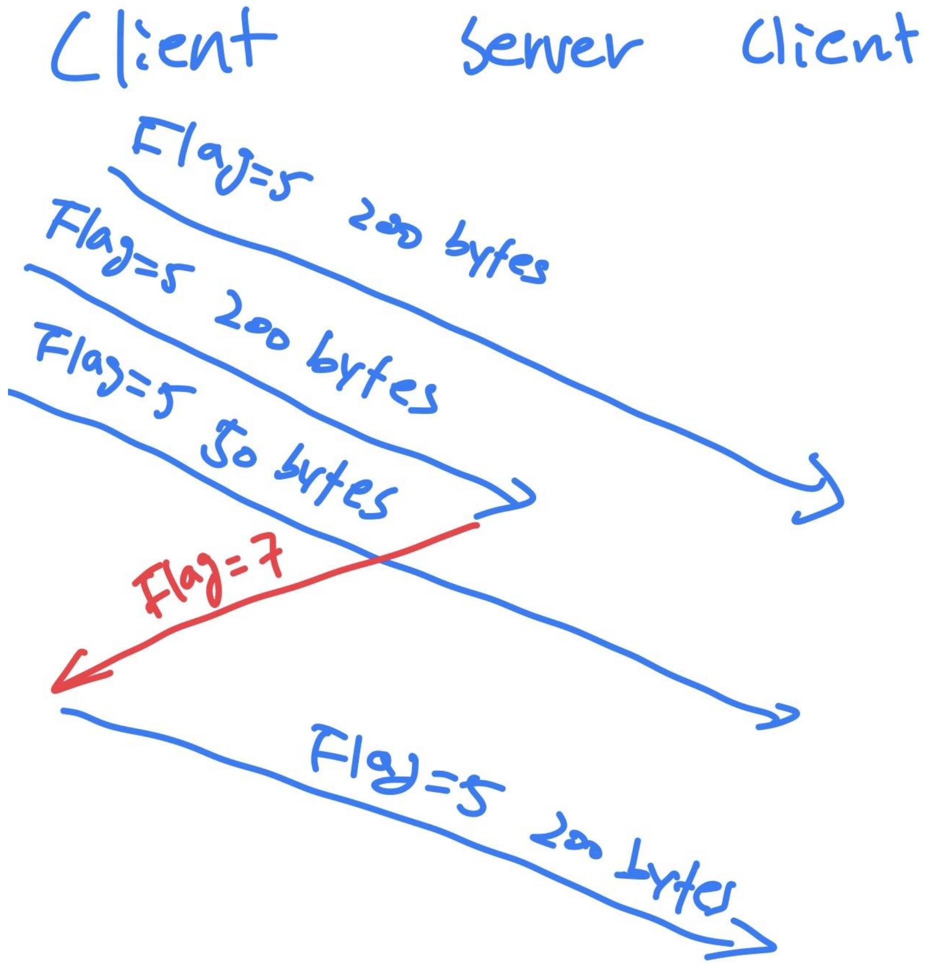
ii. **Description:** The sending client unsuccessfully sends a message to the destination clients because some (or all) or invalid handles.
**Processing on sending client:** The server sends a packet containing the message and destination client handles with the flag set to 0x5.
**Processing on server**: The server forwards the packet to all the destination clients specified in the packet. The server sends a packet containing the invalid client handles to the sending client with the flag set to 0x7.
**Process on sending client:** The client writes an error message, specifying the invalid client handles.



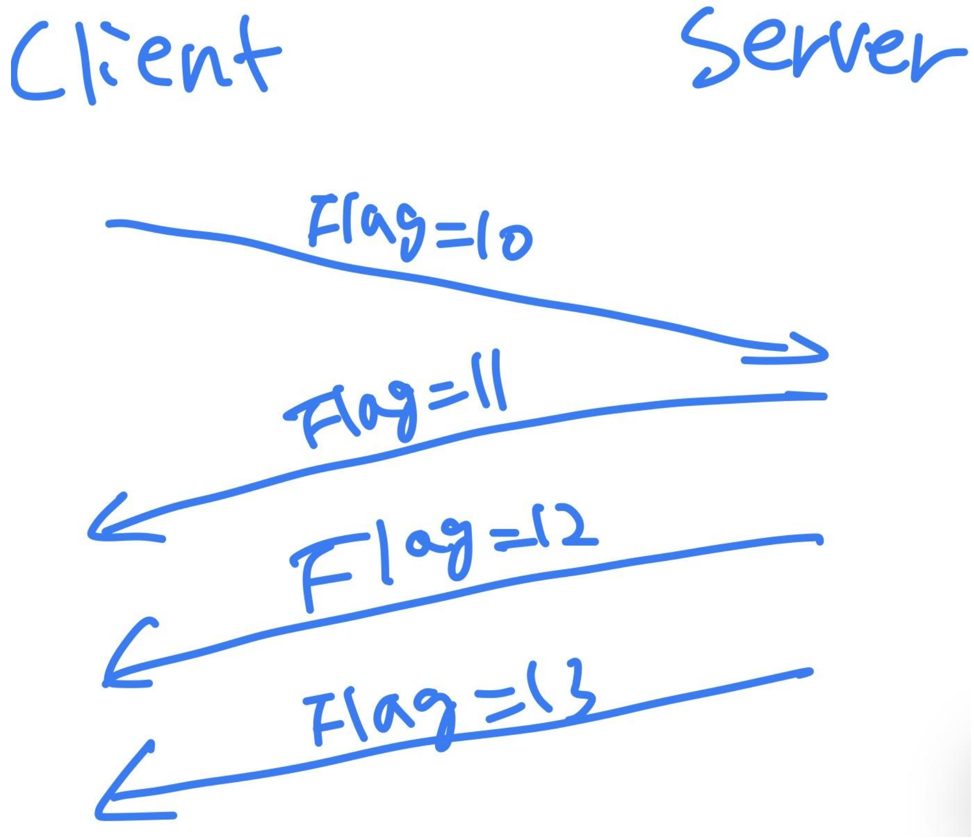
d. Broadcasting a message (%B)
**Description:** The sending client broadcasts a message.
**Processing on sending client:** The client sends a message containing the message with the flag set to 0x4.
**Processing on server:** The server forwards the message to all connected clients.
**Process on destination client:** The client writes the message to STDOUT.


Client
Server
Every Client
Flag = 4
Flag = 7
Flag = 4

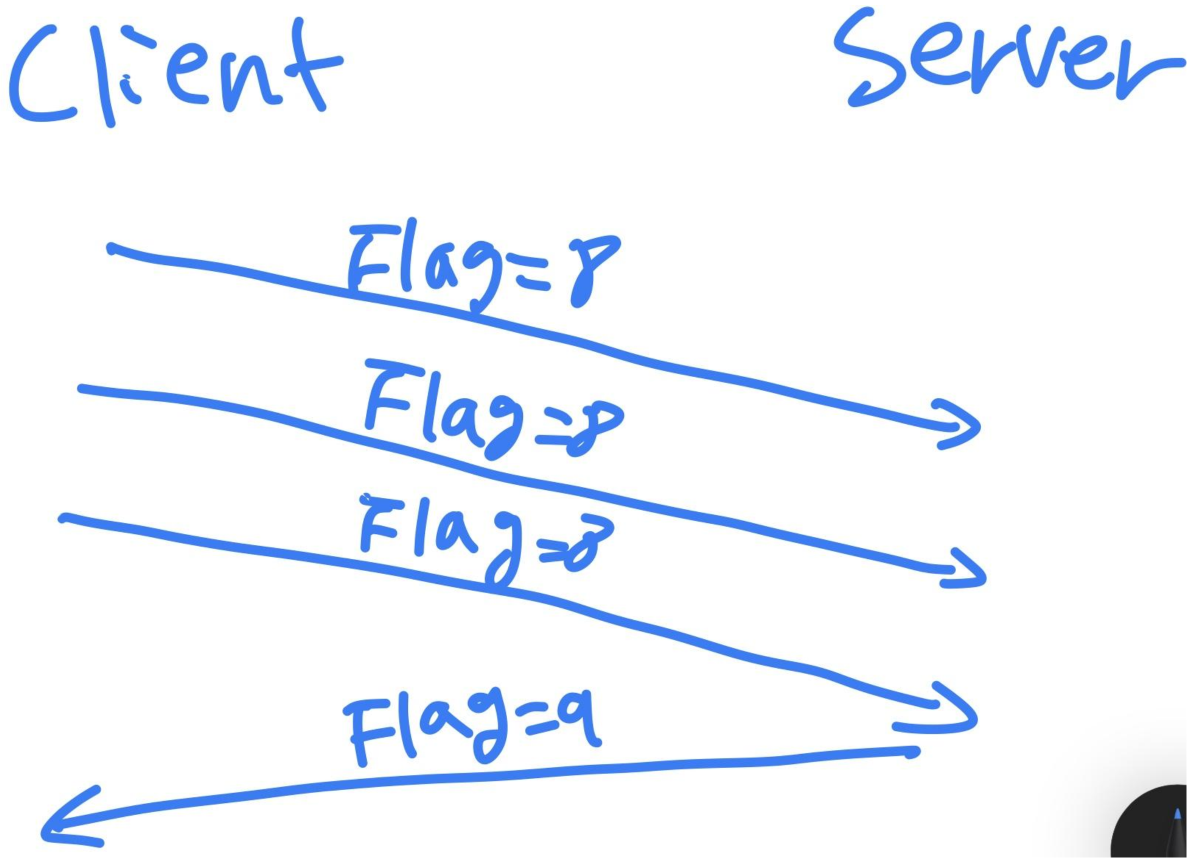
e. Sending messages if the entered message text is 450 characters
**Description:** The client sends multiple packets to encapsulate a message that is greater than 200 bytes.
**Processing on sending client:** The client sends three different packets, two containing 200 bytes of the message and the last containing 50 bytes of the message; they all have the same header.
**Processing on server:** The server forwards each of the packets to the destination client.
**Processing on the destination client:** The client writes each received message separately to STDOUT



f. Listing the handles (%L)
**Description:** The client requests a list of connected clients.
**Processing on client:** The client sends a packet with the flag set to 0xA.
**Processing on server:** The server sends a packet containing the number of handles and the flag set to 0xB, immediately followed by a series of packets each containing the handle and the flag set to 0xC, before finally sending a packet with its flag set to 0xD.
**Processing on client:** The client blocks until the packets with flag set to 0xD is received and prints out the handles to STDOUT.



g. Ending a connection (%E)
**Description:** The client ends its connection to the server.
**Processing on client:** The client sends a packet with the flag set to 0x8.
**Processing on server:** The server sends a packet with the flag set to 0x9.
**Processing on client:** The client terminates.


Client
Server
Flag=8
Flag=8
Flag=8
Flag=9

## Handle Table data structure on the server

a.

| Socket/Handle State on the server | How Can this happen (what can cause this on the server) | If you looked at your socket/handle table, how do you know the socket/handle is currently in this state (what is your data structure would tell you this) |
|---|---|---|
| Client socket is opened but handle not valid | A request is sent to the server with a misspelled or mistyped client handle. | Traversing through the linked list of handle/port mappings would reveal that the handle doesn't exist. |
| The client Socket is open and the client handle is valid. | A request is sent to the server with a correctly spelled/typed client handle. | Traversing through the linked list until a node with the requested client handle is found; its existence in the table indicates that not only does this client handle exist, but that the connection is also open. |
| The socket was opened and had a handle, but then the socket was closed | A request is sent to the server with a correctly spelled/typed client handle that is no longer connected to the server. | While the client handle may have existed, because the connection is no longer open, the entry would have been deleted from the table. As a result, the requested client handle won't be found traversing through the linked list, revealing that the client handle doesn't have a socket mapped to it. |

b.

i. addHandleSocketMapping()
   1. Inputs: a pointer to the table, a char pointer (handle), and an integer (port number)
   2. Action: Adds a new handle-socket mapping to the table.
   3. Output: void

ii. removeHandleSocketMapping()
   1. Inputs: a pointer to the table, a char pointer (handle)
   2. Action: Removes a handle-socket mapping from the table.
   3. Outputs: void

iii. getHandleSocketMapping()
   1. Inputs: a pointer to the table, a char pointer (handle)

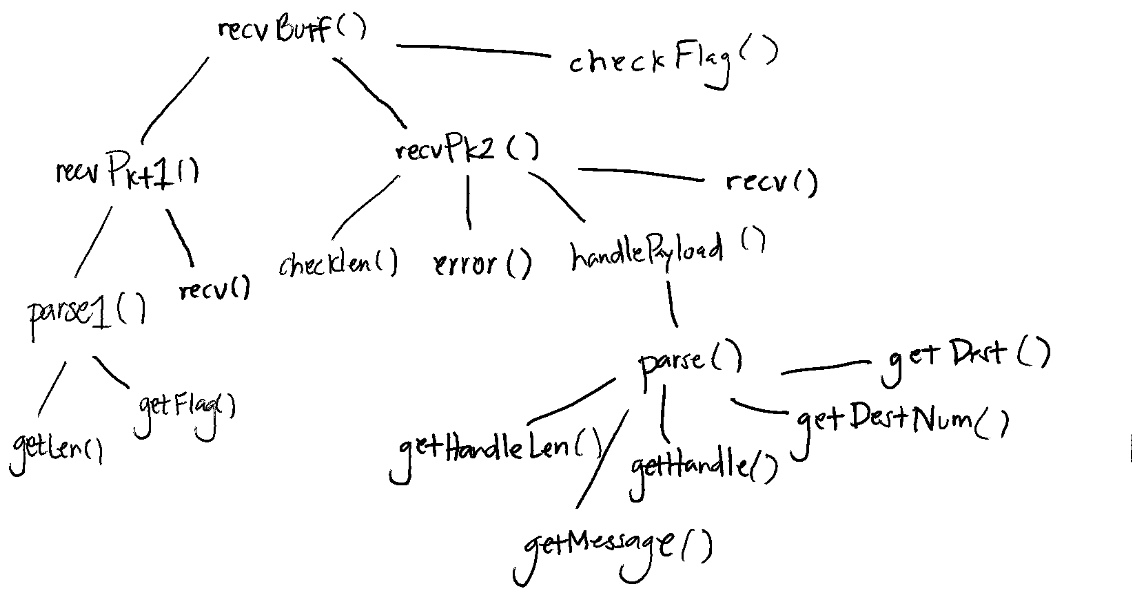
    2. Action: Get the port number mapped to the given client handle
    3. Outputs: integer (port number)

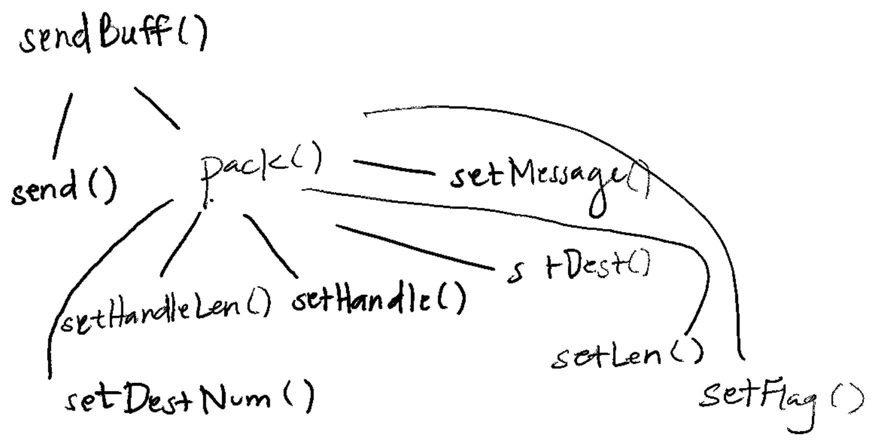
iv. changeHandleSocketMapping()

    1. Inputs: a pointer to the table, a char pointer (handle), an integer (port number)
    2. Action: Change the handle associated with the given port number to the given handle.
    3. Outputs: void

## Part 2:

### Hierarchical Drawing:





## Common Code:

The helper functions common between the two programs (client and server) from the perspective of the shared recvBuff() function are recvPkt1(), recvPkt2(), getLen(), getFlag(), getDest(), getDestNum(), getHandleLen(), getHandle(), getMessage(), checkLen(), checkFlag(). All of these are helper functions that are used to parse up and process packets. The remaining functions and their implementations, such as parse() and handlePayload() will vary based on the set flag.

The helper functions common between the two programs (client and server) from the perspective of the shared sendBuff() function are setMessage(), setHandleLen(), setHandle(), setDest(), setDestNum(), setLen(), and setFlag(). The remaining functions and their implementations, such as pack() will vary depending on the set flag.