**Implementation:**

* From the global view of this program, mainly we have 3 classes in total which represent the three specific layers in our protocol. Moreover, these 3 layers are under control of a steering method which includes a non-stop ‘while’ loop. For every loop, the program listens to either the sending part or the receiving part and do different things when entering one of them. For example, when FD (file descriptor) is ready, the procedure sends all the packets in packet\_list and keeps the packet state unless it is well delivered while when NET is ready the procedure checks the destination to do whether forwarding or receiving.
* This project starts with a very simple reliability layer with only a one-byte header which contains one bit for acknowledgement and 7 bits for the randomly generated message\_id. The simple\_reliability layer which in the code is wrapped as a class holds several functions together such as initiation, encapsulation, decapsulation and send\_ack. The idea of classification is the fundamental idea for designing rest layers in the protocol.
* The Full\_reliability layer focuses on the global unique packet number, segmentation and acknowledgement which is a more advanced and smarter version of simple\_reliabilty. In regarding to global packet number, a randomly generated 8-bit long message id and a continuous sequence number from 0 to 255 make up the global unique number. Segmentation deals with the total input and split them into one single packet every 94 bytes (the other 6 bytes are left for the complete header). It is in this layer that decides how many packets are the chatroom going to send. Function send\_ack swaps the destination and source host number in the packet and changes the acknowledge flag to 1 and repack the packet again.
* In order to do forwarding, three hard-coded dictionaries are initialized in class Forwarding which are next\_hop\_dict, lookup\_dict, reversed\_lookup\_dict respectively. Every individual host has its own version of the three dictionaries because in this static routing network each host has different neighbour hosts. Basically, the forwarding header consists of a source host number and a destination host number. The encapsulation function in this layer shift the source number to left 4 times in binary first to create the higher 4 bits in the header, then the result will concatenate with the destination host number in order to make up the 8 bits long header in this layer. Function next\_hop figures out the next neighbour this host is going to send to to reach one step closer to the destination by referring to the next\_hop\_dict.
* The very beginning of the packet header goes the checksum header. In fact, checksum header has two parts (checksum value and length of message in the packet) and each has a length of one byte. The checksum starts from summing up the binary number of the message byte by byte and then do modulo of 256 to fetch the last 8 bits as the checksum value.

**Testing and debugging:**

* In this phase, we tested our program layer by layer and then combine them together and finally achieved the implemented part working across two groups.
* Firstly, a droprate of 0.2 is brought into our program to test if the Full\_reliability does its work and successfully resends the dropped packets between two adjacent hosts (host2 & host3) in our network diagram. At first, the multiple packets message could not be delivered properly. Gradually, we found the issue is that **os.read(fd,100)** restrains the length of the message. After changing it to **os.read(fd,300),** the multi-packet message flies to its destination correctly. Besides, a bug that the host will resend all the previous unsent packets is addressed by initialising the packet list when a new message comes.
* Secondly, a third host (host9) is created to gain the result of forwarding layer. We have quite a lot attempts to achieve forwarding but fails in different situations. One major problem is that the so-called ‘router’ does not forward our message to its destination and sends destination acknowledgement back to destination host. Later, we found that we mess up the destination number and source number in the header and meanwhile the unreasonable control strategy also counts to this bug. The right of control strategy is supposed to check first if the message needs to be forwarded and then either enter next loop (means it is a ‘forwarding’ message and do nothing but jump to next loop) or do receiving part and check if it need to send back acknowledgement. In a word, ‘routers’ shall only do forwarding and do no more things on swapping the source and destination host number.
* Last but not least, sometimes (especially when packets drop happens) the checksum value does not match the correct value. The inconsistent checksum value issue is handled by initializing the packet list (discussed in previous section). Moreover, when testing between two groups, the different length of message encapsulated in the header eventually leads to message unaccepted. Finally, we agreed on using the exact length of the message as the number in header and this bug is fixed as well.