CS342: Computer Networks Lab
(July-November 2022)

Assignment-3: Reliable Transport

[Code execution and testing guide]

- **Compiling student's code in one Terminal**
  1) Run "make" command in the folder containing the *reliable.c* file
  2) Make will create ***reliable*** binary file which need to be executed using the following command.
     ./ reliable -w 5 6666 localhost:5555
     -- here "***-w***" option sets the window size to **5**. "***6666"*** is the port on which the current process is listening and "**localhost:5555**" is the address of the other client it's trying to communicate

- **Compiling testing code in other Terminal**
  1) Copy the reliable.c file in the "Testing" folder.
  2) Run "make" in the "Testing" folder.
  3) Make will create ***ref*** binary file which need to be executed using the following command.
     ./ref -w 5 5555 localhost:6666
     -- here "***-w***" option sets the window size to **5**. "***5555"*** is the port on which the current process is listening and "**localhost:6666**" is the address of the other client it's trying to communicate
  4) For the "ref" program, other options that can be used for testing are "-h" for checking the header length, and "-k" for modifying the content of a packet to test error handling.

Testing code
Your code should pass the following tests. These tests correspond to the requirements mentioned in the assignment document (the specific requirement no. are mentioned in parenthesis)

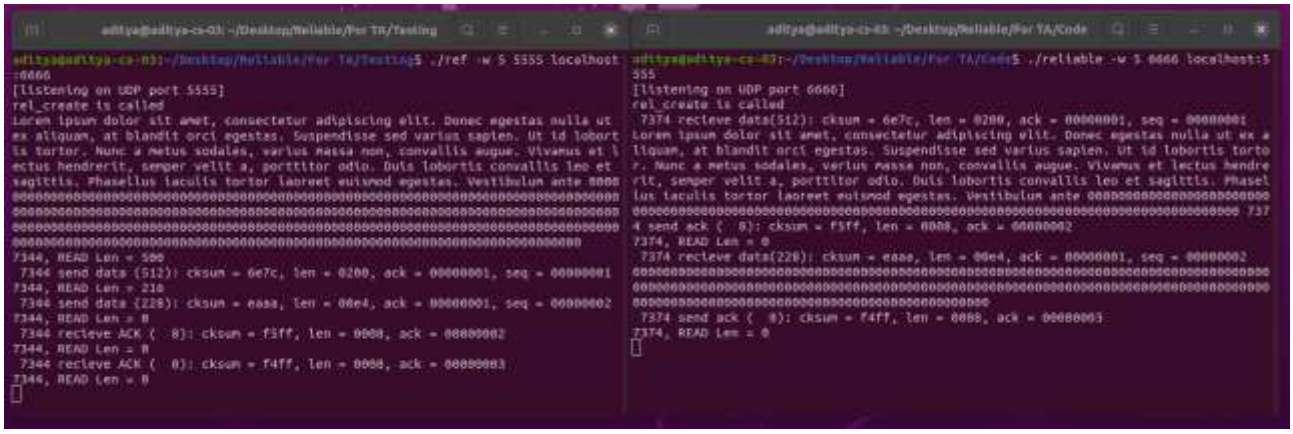
1) Testing of sending and receiving functionalities *(Requirement no. a, b (first part))***:**

Following text input in either of the client's terminal should generate two packets that are sent to the other client.

Example input:
"Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec egestas nulla ut ex aliquam, at blandit orci egestas. Suspendisse sed varius sapien. Ut id lobortis tortor. Nunc a metus sodales, varius massa non, convallis augue. Vivamus et lectus hendrerit, semper velit a, porttitor odio. Duis lobortis convallis leo et sagittis. Phasellus iaculis tortor laoreet euismod egestas. Vestibulum ante 00"

For this input, the **sender** should send two DATA packets with **sequence numbers** 1 & 2 (if it is the first message to send, otherwise sequence number will be the next integer after the last sent packet's sequence number),
and the **receiver** should send 2 ACK packets with **ack no.** 2 & 3 (if it is acknowledging the first data sent, otherwise, ack no. will be seq no. + 1 for each packet)

Example Output:



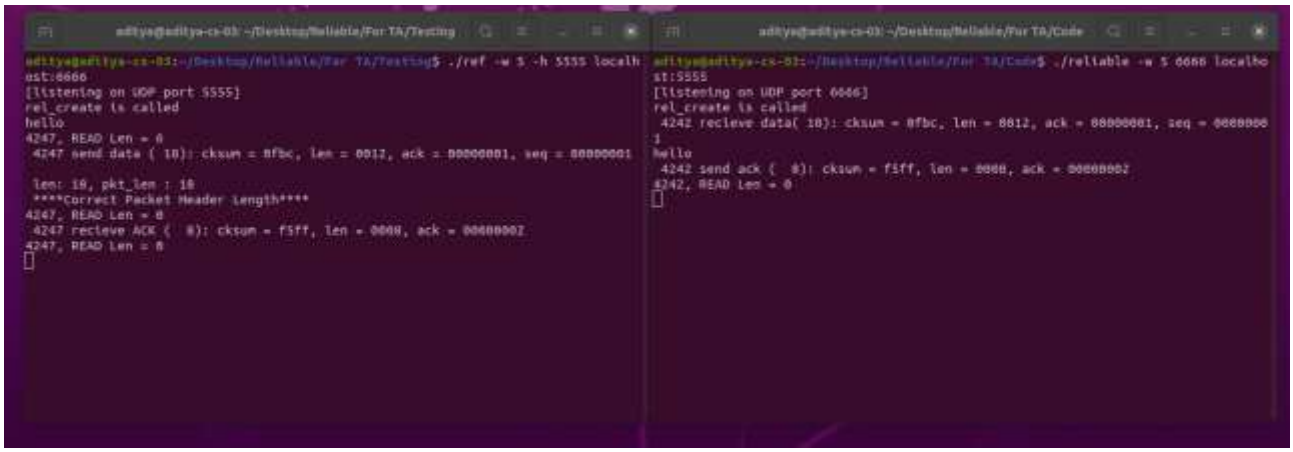
2) Checking for packet header length *(Requirement no. c (2nd part))*

Execute the **"ref"** code with the **"-h"** option to test if the **header length** is properly handled in the implementation. The **ACK packet** should have length 8 and **DATA packet** should have length greater than or equal to 12. The packet's length argument to *conn_send()* should be equal to the value inside ***packet.len*** (it should be in big-endian order. In big-endian machines, first byte of binary representation of the multi-byte data-type is stored first in the memory location, next byte is stored in next memory location, and so on.).

The first data sent from the **"ref"** client is checked for the proper header formatting. When the first packet sent by the **"ref"** client is a data packet, the header length of that packet should be 12 and otherwise the header size for ack packet should be 8.

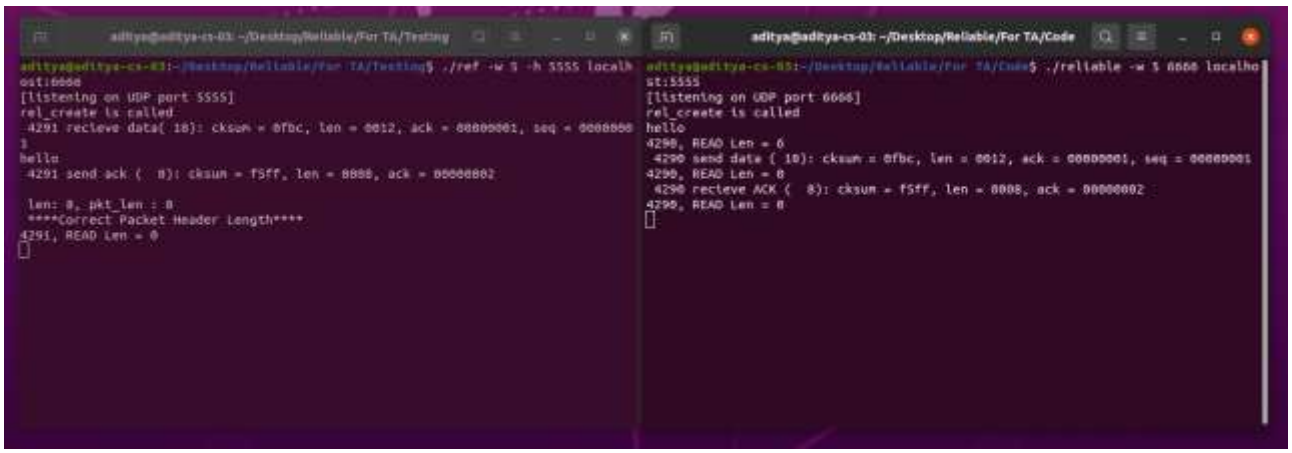
Example input:
"hello" (in the terminal running the **"ref"** client).

Example Output: <Checking header length of data packet>



Example input:
"hello" (in the terminal running the **"reliable"** client).

Example Output: <Checking header length of ack packet>



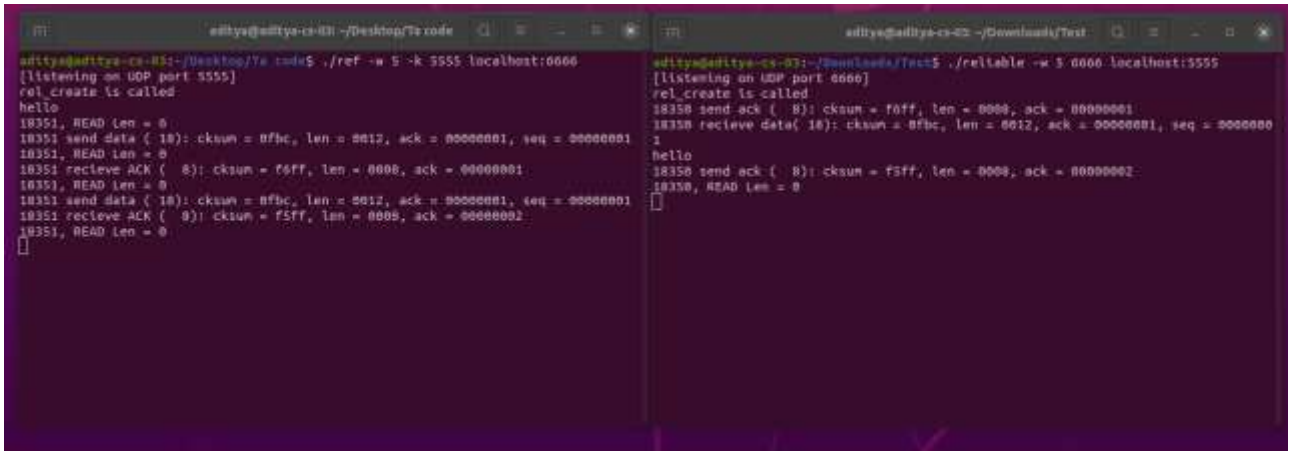
3) Checking for bit error using checksum *(Requirement no. c (1st part))*

Execute the **"ref"** code with the **"-k"** option to test the condition where the received packet's checksum is different from the value inside the ***packet.cksum*** field.

First data sent from the **"ref"** client is modified to send corrupted packet content. So, receiver should send ack packet with the seq. no. of last data packet received correctly. This ack packet should trigger the re-transmission of the original packet from **"ref"** client. This packet should now be properly acknowledged by the receiver.

Example input:
"hello"

Example Output: <Check the seq. no. of data packet. It should be repeated.>



4) Testing reliability of the implementation *(Requirement no. b(2nd part), d, e)*

To check how the code handles packet drop in this reliable transport layer implementation, we are going to use **netem**'s network emulation functionality for testing the implementation of the protocol by emulating the properties of wide area networks.

First, we need to remove any existing emulation using this command (also to remove any emulation after you are done with the testing).

sudo tc qdisc del dev lo root

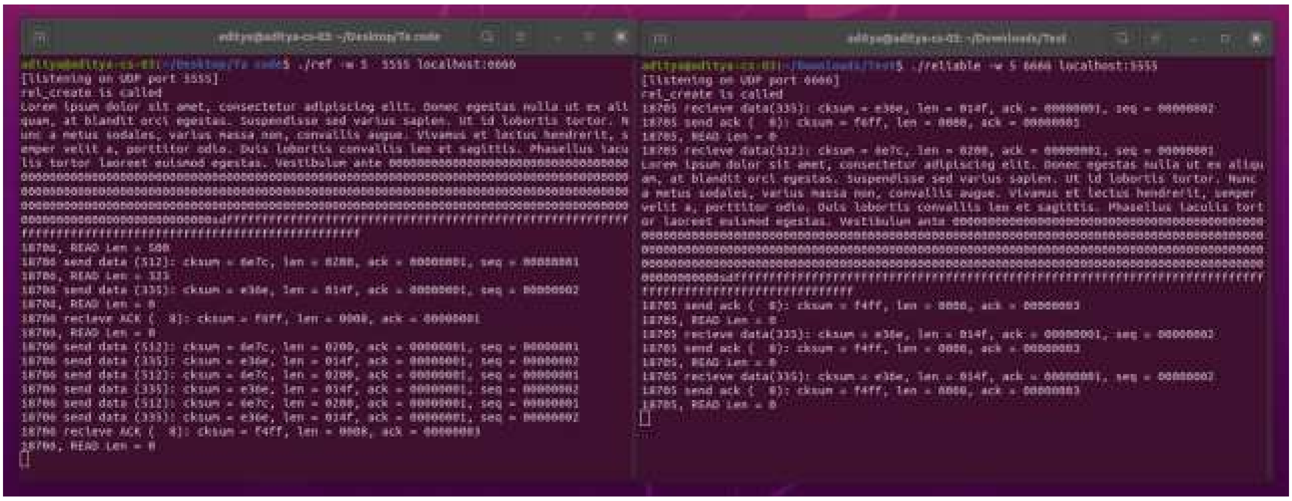
Following command can be used to add emulation random packet drop. (if you want to modify an existing emulation, use "**change"** in place of "**add**" in the below command)

sudo tc qdisc add dev lo root netem loss 50%

Using the emulation set with the commands mentioned above, 50% of the outgoing **packets will be dropped randomly**. For this test, you should check that the sender resends all outstanding data packets until they are acknowledged. And if the receiver doesn't receive the packet it's expecting, it should resend the ack packet with ack. no. of the next seq. no. you are expecting to receive.

In the example shown in the screenshot, 2 data packets were sent by the "**ref**" client, but only the second packet with seq. no. 2 was received by the "**reliable**" client, as the first packet got lost. Then the "**reliable**" client sent an ack packet with ack. no. 1 expecting to receive a packet with seq. no. 1. The "**reliable**" client ignores all other received packets having higher seq. no. than the seq. no. of the packet which is expecting to receive. When "**ref**" client receives ack packet with ack. no. 1, it starts resending all unacknowledged packets starting from seq. no. 1 (i.e., packets with seq no. 1 & 2 in this example) until it receives ACK for them. Once the "**reliable**" client receives the data packets with seq. no. 1 & 2, it sends ack packet with seq. no. of next expected packet as the ack no. i.e. ack no. 3. This ack number **shows cumulative ACK** has been followed.

Example output:



5) Testing packet reordering *(Requirement no. f)*

We are going to use **netem** for this test too.
Considering you have already set the packet-loss emulation, we are going to change that with the following command (if no emulation is set yet, use "**add"** in place of "**change**" in the below command).

sudo tc qdisc change dev lo root netem delay 100ms 20ms distribution normal

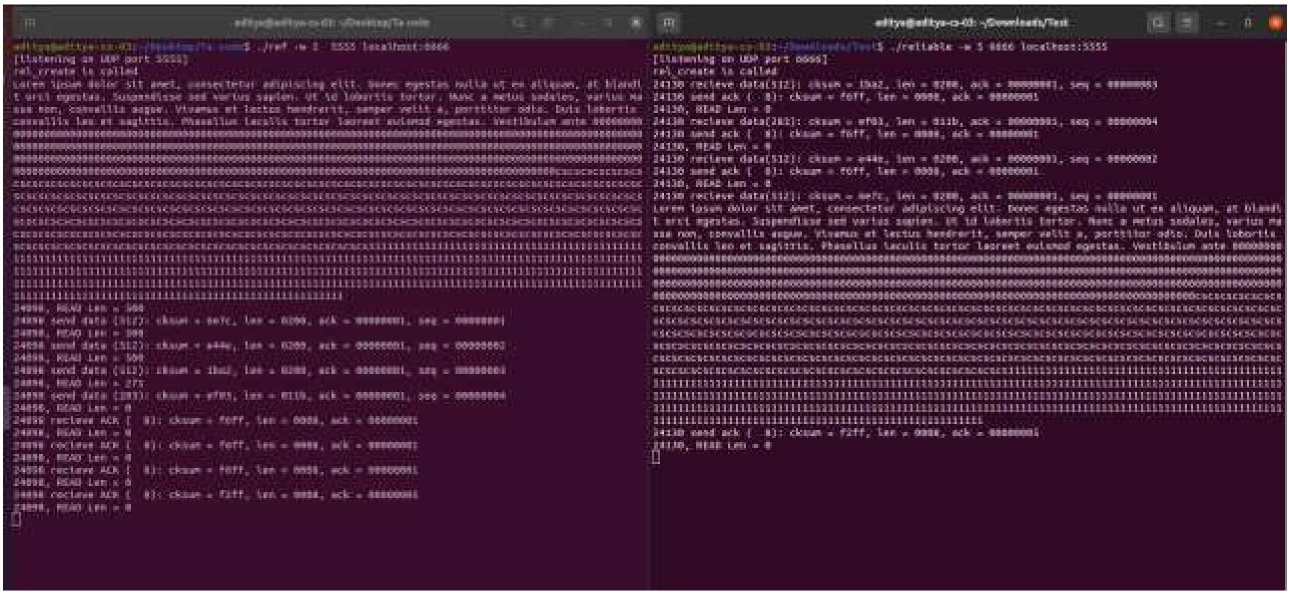
With this emulation, the delivery of the outgoing packets may experience a delay of 100ms (±20 ms) with normal distribution. This test is used to check if the data in the packets received should be printed in the order it was sent by the sender, not by the order in which it was received**.**

Example Input:
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec egestas nulla ut ex aliquam, at blandit orci egestas. Suspendisse sed varius sapien. Ut id lobortis tortor. Nunc a metus sodales, varius massa non, convallis augue. Vivamus et lectus hendrerit, semper velit a, porttitor odio. Duis lobortis convallis leo et sagittis. Phasellus iaculis tortor laoreet euismod egestas. Vestibulum ante
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000cscscscscscscscscscscscscscscscscscscscsc
scscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscs

cscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscsc
scscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscs
cscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscsc
scscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscscs
cscscs111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111

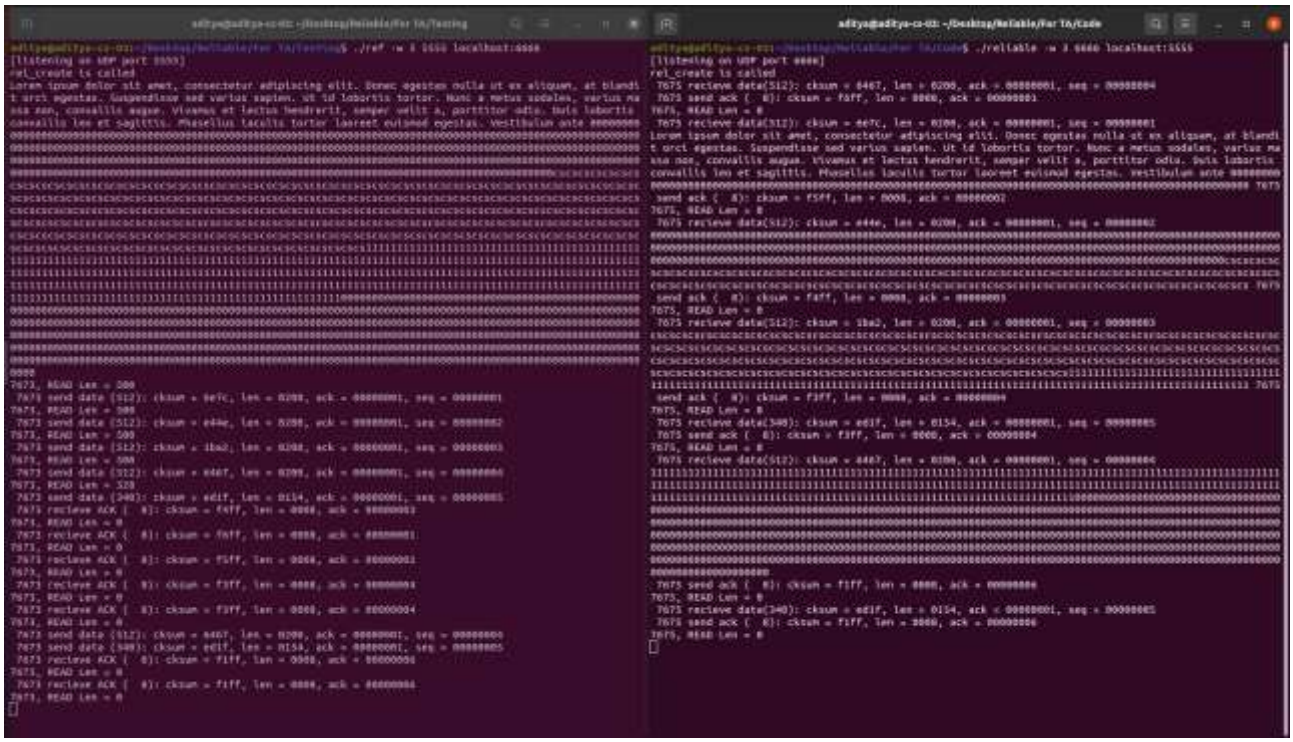
In the example below, the packets are received out of order at the receiver's end. The receiver keeps these out-of-order packets in its buffer and sends an ack packet with ack no. of the next expected data packet. The received data is printed by the receiver in the order they were sent by the sender.

Example Output:



When the receiver's window and sender's windows are of the same sizes then all unacknowledged data packets are buffer and are not requested again by the receiver. But If the window size of one of the clients is smaller than the other, the received packet may be dropped if the window is not big enough to store it. The example below shows such a scenario where the "**reliable**" client's window size is too small to fit the packet with seq. no. 4, that is delivered out-of-order. The "**reliable**" client requests for the packet with seq. no. 4 when it has space in its window again. Like in the above example, where windows sizes were the same, the receiver will not again request (by sending an ACK packet with that seq. no.) for already received packets. **Check, for correctly implemented window sizes, this case should not occur.**

Expected Output:



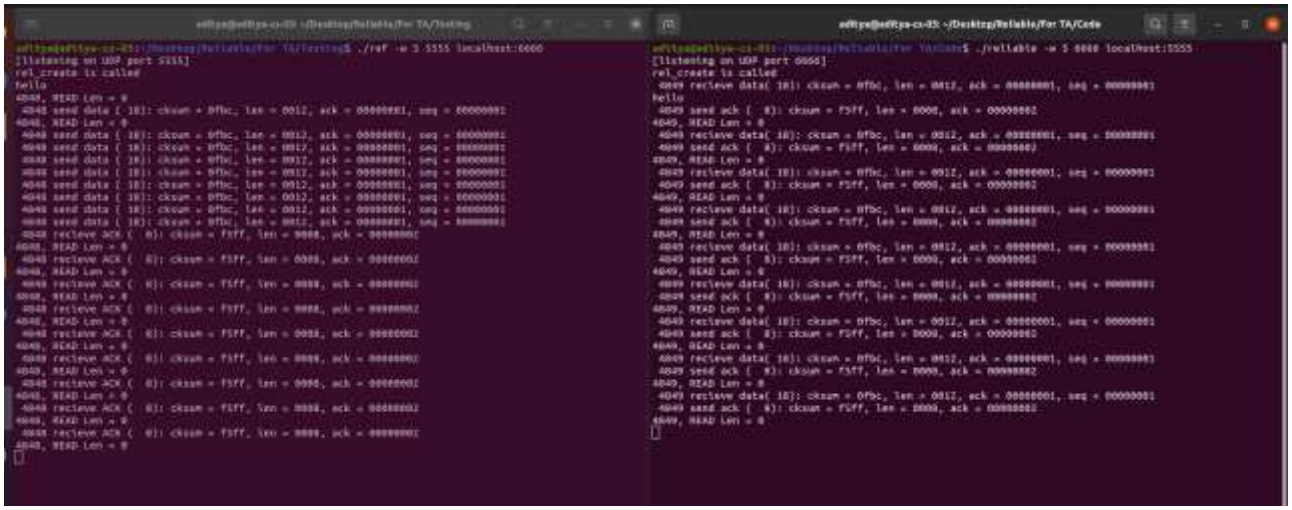
6) Testing packets resending *(Requirement no. g, h)*

Considering you have already set an emulation, we are going to change that with the following command (if no emulation is set yet, use "**add**" in place of "**change**" in the below command).

sudo tc qdisc change dev lo root netem delay 8000ms 20ms distribution normal

With this emulation, the outgoing packets will be delayed by 8 seconds and the default timeout is 2 seconds. The sender should try to resend the packets at least 5 times if the packets are not acknowledged.

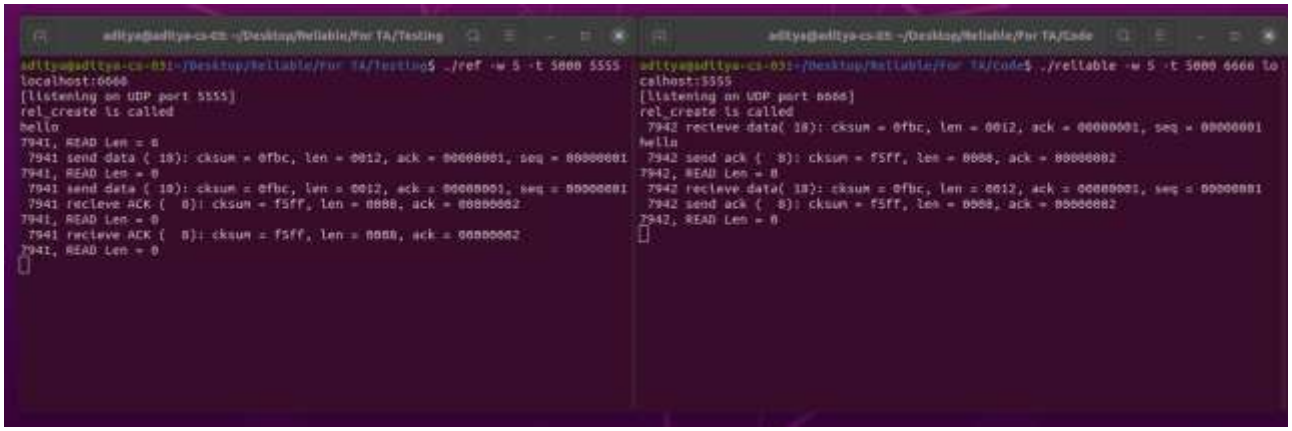
Expected Output:



In the example above, both data and ack packet experience a delay of 8 seconds, before they reach the other end. The Sender resends the data packet 8 times until it receives an acknowledgement for it.

In the example below, *timeout* is set to 5 seconds (using **-t** option) and network delay is 4 seconds (using the **netem** emulation). Since both the first data and ACK packet experience ~4 seconds delay (because of the **netem** emulation), the first ACK is received after 8 seconds, that's why the sender only re-transmits the packet once when the timer expires.

Expected Output:



7) Termination of connection *(Requirement no. i)*
When the clients from both the terminals send EOF in a data packet and all the existing packets have been acknowledged the connection should be terminated.

Input:
CTRL+D (i.e. EOF) {in both the terminals.}

Output: