Problem 1

How to Run the program

- Open two terminal
- On first terminal (server):

```
1. gcc server.c -o server
2. ./server
```

- On second terminal (client):

```
1. gcc client.c -o client
2. ./client {inputfile.txt}
```

(It takes the name of the input file as command line argument)

(e.g. If name of input file is "input.txt", run the following command: ./client input.txt)

- Output is stored in the file 'output.txt'.

Notes for running the code

Change the following parameters if required while testing in **packet.h**:

- Packet drop rate (PDR)
- server port number (PORT)
- payload size (PACKET_SIZE)
- timeout value (TIMEOUT)
- max number of tries in case of not receiving ack (MAXPENDING)

Methodology

- I used select and fd_set to handle multiple connections as required (using only one timer)
- Managing file transfer over multiple connections:

- **On the client-side**:
    - Opened two sockets with different ids and established two different connections.
    - Whenever timeout occurred, retransmit the previous packet (there are at most two unacknowledged packets as specified in Problem Statement), one for Channel 0 and other for Channel 1
    - If there was no timeout, check which ACK was received correctly. Send new packet on the channel that received the ACK correctly and retransmit the previous packet on the channel which did not receive ACK

- **On the server-side:**
    - Handled multiple clients is by using select() linux command
    - Select command allows to monitor multiple file descriptors, waiting until one of the file descriptors become active.
    - For example, if there is some data to be read on one of the sockets select will provide that information.
    - Select works as an interrupt handler, which gets activated as soon as any file descriptor sends any data.

- Data structure used for select: fd_set
    - It contains the list of file descriptors to monitor for some activity.
    - There are four functions associated with fd_set:

```
int FD_ZERO (fd_set *descriptorVector); /* removes all descriptors from vector */
int FD_CLR (int descriptor, fd_set *descriptorVector); /* remove descriptor from vector */
int FD_SET (int descriptor, fd_set *descriptorVector); /* add descriptor to vector */
int FD_ISSET (int descriptor, fd_set *descriptorVector); /* vector membership check */
```

- Created a fd_set variable readfds, which will monitor all the active file descriptors of the clients plus that of the main server listening socket.
- Whenever a new client will connect, master_socket will be activated and a new fd will be open for that client. We will store its fd in our client_list and in the next iteration we will add it to the readfds to monitor for activity from this client.
- Similarly, if an old client sends some data, readfds will be activated and we will check from the list of existing client to see which client has sent the data.

Problem 2

How to Run the program

- Open four terminals

- On first terminal (server):

```
1. gcc server.c -o server
2. ./server
```

- On second terminal (relay 1):

```
1. gcc relay.c -lm -o relay
2. ./relay 1
```

- On third terminal (relay 2):

```
1. gcc relay.c -lm -o relay
2. ./relay 2
```

- On fourth terminal (client):

```
1. gcc client.c -o client
2. ./client {inputfile.txt}
```

(It takes the name of the input file as command line argument)

(e.g. If name of input file is "input.txt", run the following command: ./client input.txt)

- Output is stored in the file 'output.txt'.

Notes for running the code

Change the following parameters if required while testing in **packet.h**:

- Packet drop rate (PDR)
- server port number (PORT)
- payload size (PACKET_SIZE)

- timeout value (TIMEOUT)
- max number of tries in case of not receiving ack (MAXPENDING)
- Relay1 port (RPORT0)
- Relay2 port (RPORT1)
- Random delay upper limit (DELAY_MAX)
- Window Size (WINDOW_SIZE)

Methodology

- Implemented a single timer per window basis

- Used select and fd_set to handle multiple connections as required

- Managing file transfer over multiple connections:

- **On the client side:**
    - Opened two sockets with different ports to establish two different UDP connections - one to relay 1 and other to relay2.
    - Whenever timeout occurred, retransmit all unacknowledged packets in the window
    - If there was no timeout, check which ACK was received correctly and mark the proper acknowledgement in window.
    - Slide the window once all the packets in a window are acknowledged.

- **On the server side:**
    - Used simple UDP send and receive to receive Data from corresponding Relay 1/2 and direct ACK to the same relay.

- **Relay**
    - Kept relay number as #define since implementation of both relay is exactly same.
    - Randomly drops pocket to cause timeout
    - Causes a delay of randomly 0 to 2000 ms for each packet using system call nanosleep()
    - Opens one socket to client and one to server

- All the logs are stored in the file 'log_file.txt'