==========
INSTRUCTION FOR COMPILING AND RUNNING PROGRAMS
==========

- To compile the codes, simple type "make" in the shell. The makefile will do the rest.

- To run the server program, in the shell, type
./kftserver <port_number> <server_to_client_loss_percent> [-d | -D]

where + <port_number>: an integer indicating the port of the server.
+ <server_to_client_loss_percent>: an integer between 0 and 100 (exclusively) indicating the loss percent when server sends a packet to the client. If this integer < 0 or > 100, the server will be terminated.
+ -d or -D: a flag to turn on the debug mode (the log of the file transfer will be printed on the screen). The debug mode is turned off by default.

- To run the client program, in the shell, type
./client <server_ip_address> <server_port_number> <remote_filename> <local_filename> <max_packet_size> <client_to_server_loss_percent> [-d | -D]

where + <server_ip_address>: a string indicating an IPv4 or IPv6 address in respective standard notation.
+ <server_port_number>: an integer indicating the port of the server.
+ <remote_filename>, <local_filename>: name of remote and local files respectively. The length of the name MUST NOT exceed MAX_LEN_FILENAME = 5000.
+ <max_packet_size>: an integer indicating the maximum segment size that UDP client and server will send and receive.
+ <client_to_server_loss_percent>: an integer between 0 and 100 (exclusively) indicating the loss percent when server sends a packet to the client. If this integer < 0 or > 100, the client will be terminated.
+ -d or -D: a flag to turn on the debug mode (the log of the file transfer will be printed on the screen). The debug mode is turned off by default.

==========
APPLICATION PROTOCOL (BUILT ON THE RELIABLE UDP)
==========
- The maximum length of a file name is 5000.
- The maximum length of a request that a client sends to the server is defined in MAX_LEN_REQUEST = 8000
- The maximum length of a packet received by the clinet is max_packet_size (given in the command line argument list).

****
Request format:
****
- A request to the server has to follow the following format:
+ The first line is the remote file name, followed by CRLF (Carriage return and line feed characters).
+ The second line is the maximum packet size (max_paket_size), followed by CRLF
+ The final line contains CRLF only which specifies the end of the request.

****
Response format:
****
- A response from the server is split into multiple packet, each of size max_packet_size.
- Because the underlying transport layer is UDP, we preserve message boundary.
- The first packet client receives has the following format:
    + The first line contains an integer indicating a status code. The first line ends with CRLF.
    + If the status code is OK = 200, there is a second line. The second line contains an integer indicating the total length of the file requested. The second line ends with CRLF
    + Rest of the first packet are data of the requested file.
- The remaining packets contains data of the requested file.
- The client knows when to stop receiving packets when the total number of bytes it has received equals the size of the file requested.

****
List of status codes currently supported and their meaning:
****
- 200: OK.
- 400: Bad request (A request that does not follow the specified syntax and format)
- 404: Not found (A requested file does not exist in the server)

==========
ARQ PROTOCOL FOR UDP
==========
- The API of rdt UDP is in rdt_udp.h. The detailed implementation is in rdt_udp.c

****
Structure of a packet (segment)
****
- Each packet / segment has size at most max_packet_size (which is specified in the command-line arguments of the client).
- The first byte contains the sequence number. Since our protocol is ARQ, the sequence number is either 0 or 1.
- The rest are data passed from the above layer.
- In rdt_client.c, there are functions:
    + make_packet: to make the packet in the specified format from the input data.
    + extract_data_from_packet: to extract data.
    + extract_seq_num_from_packet: to get the sequence number of the packet.

****
Structure of ACK
****
- In our protocol, we do use NAK. Instead, we attach the sequence number to each ACK packet to distinguish ACK and false ACK (i.e. NAK).
- The ACK packet has a sequence number as the first byte. The data of the ACK packet is the string defined in ACK_TOKEN = "ACK".
- In rdt_client.c, there is a function is_ack() to determine if a packet is an ACK packet in the specified format.

****
Sequence number synchrnization between client and server
****

- The ARQ has two main function rdt_sento() (reliable sending) and rdt_recvfrom() (reliable receiving).
- rdt_sento has access to the global sequence number variable send_seq_num.
- rdt_recvfrom has access to the global sequence number variable rcv_seq_num.
- Initially, in the client and the server, send_seq_num = rcv_seq_num = 0.
- Also, in the server, after handling a client, we reset send_seq_num = rcv_seq_num = 0. Therefore, the sequence numbers between client and server are synchronized.

****
Timeout
****
- In our implemtation, we use a fixed timeout interval.
- The interval value is defined in TIMEOUT_SECS = 1

****
RDT Sender (rdt_sendto)
****
- In this section, we present the logic of the implementation of rdt_sendto.
- rdt_sendto receives the data from the application layer. The size of the data passed to the function is specified in the application layer, and may be larger than max_packet_size. If so, the data are split into smaller chunks.
- For each chunk of data,
    + We make the packet based on the data chunk and the current value of send_seq_num.
    + Then we send the packet using UDP sendto (or in this project, dropper_sendto),
    + And we wait for the ACK from the receiver. Here some situations may occur.
    + Scenario 1: The sender receives some packet from a stranger (not from the expected receiver). In this case, we ignore that packet and continue to wait.
    + Scenario 2: Time out.
        * If the number of time we tried resending the packet exceeds MAX_TRY = 10, we give up sending and stop communicating with this host.
        * Otherwise, we resend the packet using sendto (or dropper_sendto). Also, we reset the timeout alarm and increment the number of resending times. Then we continue to wait for the ACK.
    + Scenario 3: We receive an ACK packet.
        * We check the sequence number of the packet with the current send_seq_num.
        * If they are the same, the packet has been received successfully. So we continue sending other chunks.
        * Otherwise, continue waiting for ACK.
    + Scenario 4: We receive a packet from the expected receiver, but it is not an ACK packet.
        * If the one calling rdt_sendto is the server, ignore this packet.
        * If the one calling rdt_sendto is the client, stop sending requests and move to the state of receving files from server (This is because this scenario only happens when the server receives requests, and starting executing it, but the ACK for the request is lost and the client is still waiting for it).
    + After sending each chunk successfully, we increment the send_seq_num (i.e. send_seq_num = (send_seq_num + 1) % 2).

****
RDT Receiver (rdt_recvfrom)
****
- In this section, we present the logic of the implementation of rdt_recvfrom.
- NOTE: If a packet is corrupted, UDP will drop it. Therefore, no need to check for corruption.

- Scenario 1: We receive a packet from an unexpected source (if user specify what source they expect to receive a packet from). In this case, we ignore the packet.
- Otherwise, when we receive a packet, we compare the sequence number of the packet with rcv_seq_num.
    * If they are the same, we accept the packet and send an ACK packet with sequence number equals to rcv_seq_num. And we incremnt the rcv_seq_num (i.e. rcv_seq_num = (rcv_seq_num + 1) % 2). Then we return.
    * If they are not the same, we discard the packet and send a false ACK packet with sequence number equals to (rcv_seq_num + 1) % 2. After that, we continue to wait for a packet.

==========
LIMITATION OF THE DESIGN AND PROGRAM
==========
- During the file transferring between the client and server, the final ACK can be lost. However, the client knows how to stop receiving because of the file length. But the server will keep resending until the number of resending times exceeds MAX_TRY. This is somehow inefficient in certain case. But the correctness of the protocol is guaranteed. Also, the chance the final ACK lost is low.
- When a server handles a client, if another client sends a request, then that request will be ignored. There is no queue of client sockets (like TCP) in this implementation.

==========
STRENGTH OF THE PROGRAM
==========
- Can be used to transfer any kind of files (NOT limited to text files).
- Support both IPv6 and IPv4 addresses.