Function Descriptions for du-proto.c

1. Function: dpinit
    a. Initializes a new dp_connection structure, sets initial values, and returns a pointer to it.
2. Function: dpclose
    a. Frees the memory allocated for a dp_connection structure.
3. Function: dpmaxdgram
    a. Returns the maximum size of the data gram buffer.
4. Function: dpServerInit
    a. Initializes a dp_connection structure for the server, sets up the socket, and binds it to the specified port. Returns a pointer to the initialized dp_connection structure.
5. Function: dpClientInit
    a. Initializes a dp_connection structure for the client, sets up the socket, and assigns the server address and port. Returns a pointer to the initialized dp_connection structure.
6. Function: dprecv
    a. Receives data from a dp_connection and stores it in the provided buffer. Returns the size of the received data.
7. Function: dprecvdgram
    a. Receives a raw datagram and performs basic validation. Returns the size of the received data or an error code.

8. Function: dprecvraw
    a. Receives raw data from the socket and stores it in the buffer. Returns the number of bytes received or an error code.
9. Function: dpsend
    a. Sends data through a dp_connection using the provided buffer. Returns the size of the sent data.
10. Function: dpsenddgram
    a. Sends a datagram through a dp_connection. Returns the size of the sent data or an error code.
11. Function: dpsendraw
    a. Sends raw data through a dp_connection. Returns the number of bytes sent.
12. Function: dplisten
    a. Server listens for incoming connection requests and establishes a connection. Returns true on successful connection, otherwise returns an error code.
13. Function: dpconnect
    a. Client sends a connection request to the server and waits for an acknowledgment. Returns true on successful connection, otherwise returns an error code.
14. Function: dpdisconnect
    a. Client sends a disconnection request to the server and waits for an acknowledgment. Returns DP_CONNECTION_CLOSED on successful disconnection, otherwise returns an error code.

15. Function: dp_prepare_send
    a. Prepares a buffer for sending by copying the PDU header into it. Returns a pointer to the buffer after the PDU header.
16. Function: print_out_pdu
    a. Prints the details of an outgoing PDU if debug mode is enabled.
17. Function: print_in_pdu
    a. Prints the details of an incoming PDU if debug mode is enabled.
18. Function: print_pdu_details
    a. Prints the detailed information of a PDU.
19. Function: pdu_msg_to_string
    a. Converts a PDU message type to a human-readable string. Returns a string representing the message type.
20. Function: dprand
    a. Generates a random boolean based on the given threshold. Returns 1 if the random number is less than the threshold, otherwise returns 0.

2. The top layer (dpsend(), dprecv()) manages the entire data buffer and interfaces directly with the application. The middle layer (dpsenddgram(), dprecvdgram()) handles datagram processing, including fragmentation and reassembly. The bottom layer (dpsendraw(), dprecvraw()) is responsible for the raw transmission of data over the network using UDP. This design makes the code easier to understand, debug, and maintain by breaking down the functionality into smaller, manageable parts. I think this design choice is good because each layer has a specific responsibility. This helps to separate different concerns and makes it easier to update and improve.

3. Sequence numbers are important for keeping data transmission organized and reliable. Each data packet sent or received is given a sequence number. This helps ensure that the packets are put together in the right order, even if they come in the wrong order. Sequence numbers also help the receiver find missing packets and ask for them to be sent again, making communication more dependable. Sequence numbers are crucial for handling the splitting up and putting back together of large buffers, making sure that all the parts are correctly identified and reassembled. It's important to update the sequence number when sending an ACK response because it helps keep track of the progress of communication between the sender and receiver. It also prevents the processing of duplicate packets and ensures that each packet is received and acknowledged, making the transmission reliable.

4. The 'du-proto' protocol requirement of acknowledging every send before proceeding to the next can lead to lower throughput compared to traditional TCP due to the additional latency it introduces. The protocol's design simplifies its implementation by avoiding the complexity of managing unacknowledged packets and handling out-of-order packets, which are standard in TCP. Despite the limitation, this simplicity makes the protocol easier to implement and debug, with fewer states to manage and more predictable behavior.

5. UDP is a type of connection that allows data to be sent without the need to set up a connection first. This means there is less initial setup required. However, it also means that UDP does not ensure that all data packets will arrive or be received in the correct order. On the other hand, TCP requires a connection to be established between the client and server using a three-way handshake before any data can be transferred. TCP also includes reliability and error checking features, whereas with UDP, you need to manage these aspects yourself. Another key difference between UDP and TCP that I noticed is that with UDP, you just create a socket and use sendto and recvfrom functions to send and receive datagrams. On the other hand, TCP requires establishing a connection using connect for clients and listen and accept for servers.