1. du-proto.c

```
/**
* dpinit()
*
* This function allocates memory for a new dp_connection structure, clears its
* contents, and initializes its fields.
* It sets input and output socket addresses as uninitialized, assigns their lengths as the
* size of a sockaddr_in structure, initializes the sequence number to zero, sets the
* connection status as false, and marks the debugging mode as true.
*
* It then returns the pointer to the dp_connection structure that it allocated and
* initialized.
*/

/**
* dpclose(dp_connp dpsession)
*
* This function takes a dp_connp (pointer to dp_connection structure) and frees the
* memory allocated for it to close the session.
*
*/

/**
* dpmaxdgram()
* This function returns the max datagram buffer size, which is defined by the
* DP_MAX_BUFF_SZ constant (512) defined in du-proto.h.
*
*/
```

```
/**
* dpServerInit(int port)
*
* This function takes a port number to create a server-side connection.
* It initializes a dp_connection structure, sets up a UDP socket, fills the input socket
* address fields in the dp_connection structure with server info, and sets socket
* options so local addresses can be reused.
*
* It returns a pointer to the initialized dp_connection structure on success or NULL if an
* error occurs.
*
*/

/**
* dpClientInit(char *addr, int port)
*
* This function takes the server IP address and the port number the server is listening on
* to create a client-side connection.
* It initializes a dp-connection structure, sets up a UDP socket, fills the output socket
* address fields in the dp_connection structure with the server's information (like the IP
* address and port). It also copies the outbound address to the inbound address.
*
* It returns a pointer to the initialized dp_connection structure on success or NULL if an
* error occurs.
*
*/
```

```
/**
* dprecv(dp_connp dp, void *buff, int buff_sz)
*
* This function takes a pointer to a dp_connection structure, a buffer and its size.
* It calls the dprecvdgram function to receive a datagram and put it in a buffer
* (_dpBuffer). If the connection is closed, it returns the DP_CONNECTION_CLOSED
* constant if it is true. Otherwise, it casts _dpBuffer to a pointer to dp_pdu and copies
* the payload (if there is one) into the provided buffer.
*
* It returns the payload size (dgram_sz) or DP_CONNECTION_CLOSED if the
* connection was closed.
*


/**
* dprecvdgram(dp_connp dp, void *buff, int buff_sz)
*
* This function takes a pointer to a dp_connection structure, a buffer and its size.
* It receives a datagram from the connection using the dprecvraw function and checks
* for errors, returning an appropriate error code if one occurs. If the datagram is valid, it
* copies the datagram contents into the provided buffer and updates the PDU's
* sequence number based on the dgram_sz field from the dp_pdu structure, which
* indicates the size of the pdu payload.
* Then it prepares an out pdu to send error messages if needed or to send
* acknowledgements like SNDACK or CLOSEACK depending on the message type.
*
* It returns the number of bytes received, DP_CONNECTION_CLOSED if the connection
* is closed, or an error code if an error occurred.
*/
```

```
/**
* dprecvraw(dp_connp dp, void *buff, int buff_sz)
*
* This function takes a pointer to a dp_connection structure, a buffer and its size.
* It receives a raw datagram from the UDP socket through the connection and stores
* it in the provided buffer. The connection's outSockAddr info is updated and if
* enabled, it prints the payload of the received datagram. It also prints the pdu info.
*
* It returns the number of bytes received or -1 if an error occurs.
*/

/**
* dpsend(dp_connp dp, void *sbuff, int sbuff_sz)
*
* This function takes a pointer to a dp_connection structure, a buffer and its size.
* It checks to see if the buffer size is greater than the max allowed datagram size.
* If it is not too large, the datagram is sent using dpsenddgram().
*
* It returns the number of bytes successfully sent or the error code
* DP_BUFF_UNDERSIZED if the buffer size exceeds the max allowed size.
*/

/**
* dpsenddgram(dp_connp dp, void *sbuff, int sbuff_sz)
*
* This function takes a pointer to a dp_connection structure, a buffer and its size.
* It sends a datagram through the connection, ensuring the buffer size is within limits
* and the connection is properly initialized. If any errors are found, appropriate
```

```
* error codes are returned.
* It constructs the out pdu using _dpBuffer, where the header is initialized first before
* the payload is copied from the provided buffer. The complete datagram is sent with
* dpsendraw(). After sending, the out pdu's sequence number is updated based on the
* payload size and the function waits for an acknowledgment (`DP_MT_SNDACK`) from
* the receiver.
*
* It returns the number of bytes that make up the payload or an error code if an error
* occurs.
*/

/**
* dpsendraw(dp_connp dp, void *sbuff, int sbuff_sz)
*
* This function takes a pointer to a dp_connection structure, a buffer and its size.
* It sends raw datagrams through the connection. It checks if the connection is properly
* initialized and returns -1 if it isn't. Then it uses sendto() to send data in the
* provided buffer over the UDP socket and update the outSockAddr fields in the
* connection, and the out pdu is printed.
*
* It returns the number of bytes sent or -1 if the connection wasn't set up properly.
*/

/**
* dplisten(dp_connp dp)
*
* This function takes a pointer to a dp_connection structure.
* It listens for a connection and handles the acknowledgements needed to establish
```

```
* the connection. It checks if the connection is properly initialized and returns an error if
* it isn't. Then it waits to receive a connection request with dprecvraw(). If the
* connection request was received successfully, it sends a connection
* acknowledgement through the connection using dpsendraw(). Once the connection is
* successfully established, the connection status is updated to true. If any errors occur
* during the process, such as improper connection initialization or issues with receiving
* the connection request or sending the acknowledgment, a general error code is
* returned.
*
* It returns true if a connection was established or DP_ERROR_GENERAL if an error
* occurred.
*/

/**
* dpconnect(dp_connp dp)
*
* This function takes a pointer to a dp_connection structure.
* It establishes a connection by sending a connection request with dpsendraw() and
* receives an acknowledgement message with dprecvraw(). Once the connection is
* successfully established, the sequence number is updated, and the connection
* status is updated to true. If any errors occur during the process, such as improper
* connection initialization or issues with sending the connection request or receiving
* the acknowledgment, an error code is returned.
*
* It returns true if a connection was established or  -1 if an error occurred.
*/

/**
```

```
* dpdisconnect(dp_connp dp)
*
* This function takes a pointer to a dp_connection structure and sends a close
* connection request. It initializes a pdu to send the close connection request and
* sends it with dpsendraw(). It then receives a close connection acknowledgement with
* dprecvraw() and closes the connection. If any errors occur during the process, such
* as issues with sending the disconnect request or receiving the acknowledgment, an
* error code DP_ERROR_GENERAL is returned.
*
* It returns DP_CONNECTION_CLOSED if the disconnection was successful or
* DP_ERROR_GENERAL if there were errors.
*/

/**
* dp_prepare_send(dp_pdu *pdu_ptr, void *buff, int buff_sz)
*
* This function takes a pdu pointer, a buffer, and its size.
* If the buffer size is smaller than the pdu header size, NULL is returned.
* Otherwise, the function prepares a buffer to be used for sending by copying the pdu
* header to the provided buffer after clearing the buffer.
*
* It returns a pointer to the buffer after the pdu header or NULL if there is an error.
*/

/**
* print_out_pdu(dp_pdu *pdu)
*
* This function takes a pointer to an out pdu and prints out its details if debugMode is
```

```
* enabled with print_pdu_details().
*/

/**
* print_in_pdu(dp_pdu *pdu)
*
* This function takes a pointer to an in pdu and prints out its details if debugMode is
* enabled with print_pdu_details.
*/

/**
* print_pdu_details(dp_pdu *pdu)
*
* This function takes a pointer to a pdu prints its fields neatly.
*/

/**
* pdu_msg_to_string(dp_pdu *pdu)
*
* This function takes a pointer to a pdu and returns a string message depending on its
* message type.
*/

/**
* dprand(int threshold)
*
* This function takes a threshold number and returns 0 or 1 based on the number.
* If the number is in between 1 and 99 inclusive, a random number is generated. 0 or 1
```

* will be returned based on the comparison between the random number and the

* threshold number.

*/

2. The responsibility of the top layer (dpsend() and dprecv()) is to provide an interface for the user, abstracting away the low-level details so that users don't need to understand the finer points to use the functions. The responsibility of the middle layer (dpsenddgram() and dprecvdgram()) is to add additional processing, such as building PDUs, using the bottom layer functions to send and receive datagrams, handling acknowledgements, and error checking while keeping these tasks hidden from the top layer. The bottom layer (dpsendraw() and dprecvraw()) handles the actual data transmission using C's built-in functions, directly interfacing with the socket for network communication. This is a good design because it clearly separates responsibilities, making each layer modular and easier to update or reuse without impacting the others.
3. In the du-proto protocol, sequence numbers track the order of datagrams as they are sent and received. This is so the receiver can process them in the correct order and see if there is a missing or out-of-order datagram. Sequence numbers are updated for things that must be acknowledged to confirm that the datagram was received. Without this, the sender wouldn't know if a datagram needs resending if it goes missing.
4. This approach is slower than TCP because it requires waiting for an acknowledgment before sending, unlike TCP, which can send multiple packets at the same time. However, it simplifies the implementation of the protocol since there isn't a need for complicated flow control or dealing with network congestion.
5. For UDP, the socket setup and management are simpler compared to TCP because UDP is stateless. Each datagram is sent without requiring a connection or managing a state. Unlike TCP, UDP doesn't need the three-way handshake or connection management. A UDP socket needs the destination IP and the destination port while TCP needs the source IP, destination IP, source port, and destination port. Also, since UDP is not reliable and does not handle errors like TCP does, it is faster and simpler to set up.