**1. Function Descriptions for du-proto.c**

dpinit()

Allocates and initializes a dp_connection structure, setting up the initial state for either a client or server connection. This includes setting address initialization flags to false, sequence number to zero, and enabling debug mode. The function is foundational for establishing a new Drexel Protocol session, ensuring that the connection starts with a clean state. It's crucial for both dpServerInit and dpClientInit functions as they rely on this initial setup.

dpServerInit(int port)

Prepares a server-side connection by creating a UDP socket, configuring it with the specified port, and applying socket options to allow address reuse. It initializes server address structures and binds the socket to the specified port, marking the server ready to listen for incoming connections. This function encapsulates server-specific initialization, including socket creation and binding, which are essential steps in setting up a UDP server. The use of SO_REUSEADDR allows the server to quickly restart without waiting for the socket to close.

dpClientInit(char *addr, int port)

Initializes a client-side connection by creating a UDP socket and setting up the server's address and port for future communications. It prepares the client to send data to the specified server address. Essential for client-side operations, this function sets the groundwork for a client to communicate with a UDP server by specifying where messages should be sent.

dpsend(dp_connp dp, void *sbuff, int sbuff_sz)

Public interface function for sending data. It encapsulates the data in a protocol data unit (PDU) and sends it to the connected peer. Handles protocol-specific wrapping of application data. Serves as the high-level entry point for sending data, ensuring that data packets are correctly formatted according to the du-proto protocol specifications before transmission.

dprecv(dp_connp dp, void *buff, int buff_sz)

Public interface function for receiving data. It waits for incoming data, extracts the payload from the protocol data unit (PDU), and delivers it to the application. As a counterpart to dpsend, this function is vital for receiving and processing incoming data packets, stripping protocol headers, and handling errors or control messages accordingly.

dp_prepare_send(dp_pdu *pdu_ptr, void *buff, int buff_sz)

Prepares a buffer for sending by copying the PDU header into the buffer before appending actual data. It's used internally to structure data packets correctly. Facilitates the construction of outgoing data packets by ensuring that the PDU header precedes the application data, maintaining protocol integrity.

pdu_msg_to_string(dp_pdu *pdu)

Converts message type codes from the PDU structure into human-readable string representations. Useful for debugging and logging purposes. Enhances the readability of protocol operations by providing clear textual representations of various message types, aiding in troubleshooting and protocol analysis.

dprand(int threshold)

Generates a pseudo-random decision based on a specified threshold, intended for simulating network conditions or errors in a controlled manner. While not directly related to the core protocol functionality, this utility function is valuable for testing error handling, simulating packet loss, or injecting faults for robustness evaluation.

dpsenddgram(dp_connp dp, void *sbuff, int sbuff_sz) and dprecvdgram(dp_connp dp, void *buff, int buff_sz)

These functions implement the protocol-specific logic for sending and receiving datagrams, respectively. They handle the preparation and processing of PDUs, including setting message types and sequence numbers. They bridge the gap between the high-level API and the low-level network communication, encapsulating the du-proto specific behaviors such as packet formatting, sequence numbering, and acknowledgment processing.

dpsendraw(dp_connp dp, void *sbuff, int sbuff_sz) and dprecvraw(dp_connp dp, void *buff, int buff_sz)

Directly handle sending and receiving data over the network without applying any protocol-specific logic. These functions interact with the UDP socket to transmit and receive raw data packets. At the core of network communication, these functions are responsible for the actual data transfer over the network. They ensure that data reaches.

dpclose(dp_connp dpsession)

Frees the resources associated with a dp_connection structure and effectively closes the UDP socket, ending the session. This function is crucial for resource management and cleanly terminating protocol sessions. It ensures that resources allocated during the connection initialization (e.g., memory for dp_connection) are properly released, preventing memory leaks and ensuring the socket is closed to avoid port exhaustion issues.

dpmaxdgram()

Returns the maximum data size (in bytes) that can be sent in a single datagram, as defined by DP_MAX_BUFF_SZ. This value is critical for enforcing the limits of protocol data units (PDUs) and ensuring compatibility across different network configurations. The function abstracts the concept of maximum transmission unit (MTU) handling within the protocol, allowing for easy adjustments and providing a clear boundary for application developers using the protocol.

dplisten(dp_connp dp)

Prepares the server to listen for incoming connection requests. It blocks until a dpconnect request is received, then sends back a connection acknowledgment, setting the foundation for a communication session. This function embodies the server-side logic for establishing protocol-based connections over UDP, which traditionally doesn't support connections. It simulates a connection establishment phase by handling specific protocol messages.

dpconnect(dp_connp dp)

Initiates a connection from the client side by sending a connection request to the server and waiting for an acknowledgment. This function mimics the connection establishment phase of connection-oriented protocols like TCP, adapted for UDP. It showcases how connection-oriented behavior can be implemented on top of a connectionless protocol (UDP) by using custom protocol messages and waiting for server acknowledgments before proceeding.

dpdisconnect(dp_connp dp)

Handles the disconnection process for the client, sending a close connection message to the server and awaiting acknowledgment. This ensures both parties are aware of the session termination. This function is essential for gracefully ending communication sessions, preventing abrupt disconnections and allowing resources to be freed correctly on both client and server sides.

print_out_pdu(dp_pdu *pdu) and print_in_pdu(dp_pdu *pdu)

These functions are designed for debugging purposes, providing formatted output of the details contained within a Protocol Data Unit (PDU) structure. They differ in their focus, with print_out_pdu targeting outgoing PDUs and print_in_pdu for incoming PDUs. By offering insights into the content of PDUs as they are sent and received, these functions are invaluable tools for developers to understand the protocol's operation and troubleshoot issues. They contribute to the protocol's transparency and ease of debugging.

print_pdu_details(dp_pdu *pdu)

Works closely with the print_out_pdu and print_in_pdu functions to format and display the contents of a PDU, including version, message type, message size, and sequence number. It decodes and presents these details in a human-readable format. This function plays a critical role in debugging and protocol analysis by breaking down the PDU fields, aiding in the quick identification of issues related to protocol operations, such as unexpected message types or sequence number mismatches.

Each of these functions plays a vital role in the du-proto protocol's operation, from managing connections and session lifecycle to aiding in debugging and protocol analysis.

**2. Sublayers of Send and Receive:**

*For Send:*

dpsend(): The dpsend() function is designed to send data from the client to the server. It only checks for if the data sent has size larger than limit then leverages two sub-layer functions: dpsenddgram() and dpsendraw().

dpsenddgram(): Prepares the data packet for transmission by wrapping the application data in a custom Protocol Data Unit (PDU) structure. This involves setting the correct protocol version, message type, sequence number, and the size of the data being sent. It then copies the application data into a buffer right after the PDU header. Then, it expects receiving an ACK Message.

dpsendraw(): Takes the prepared data packet from dpsenddgram() and sends it over the network using the UDP socket. It directly interacts with the underlying transport layer, handling the actual transmission of bytes.

*For Receive:*

dprecv(): The dprecv() function is responsible for receiving data on the server side. It checks for Connection closed and buffers storing the data then relies on dprecvdgram() and dprecvraw() for its operation.

dprecvdgram(): Waits for incoming data packets, extracts the PDU header to interpret the packet according to the protocol, and handles acknowledgment logic. It checks for errors, acknowledges received packets by sending ACKs back to the sender, and updates the sequence number based on the received data. Then, it sends back ACK based off the Message just received. (If it is CLOSE ACK then it closes the connection)

dprecvraw(): Directly interacts with the UDP socket to receive raw data from the network. It deals with the nuances of network communication, such as handling the socket flags and extracting the sender's address.

The distinct separation between handling application data, protocol logic, and raw network communication fosters modularity. This design makes it easier to modify or extend each layer independently, such as updating the protocol logic without altering the network communication code. By isolating the raw send and receive logic into separate functions, these operations can be reused across different parts of the application or in different protocols that might run on top of UDP, reducing code duplication. Also, the clear division of responsibilities enhances code readability (much easier to understand) and maintainability. The current design does have some rooms for improvements: integrating retry mechanisms for failed sends and more sophisticated acknowledgment handling could further enhance protocol robustness, incorporating parallel processing techniques to handle multiple sends/receives concurrently, have dynamic window sizing, integrating encryption and authentication mechanisms at the appropriate layer to address security concerns.

**3. Sequence Numbers in du-proto**

Sequence numbers in du-proto are used to track the order of messages sent and received, ensuring reliable data transfer. Each time a message is sent or received, the sequence number is incremented with the message size, which is crucial for acknowledging received data. Updating the sequence number for acknowledgments (ACKs) ensures both sides of the connection maintain a consistent view/synchronization of the data transfer state, allowing for proper flow control and loss recovery mechanisms.

**4. Limitation and Advantage of Acknowledgment Requirement Before Next Send**

Requiring every send to be acknowledged before the next send introduces significant latency, especially in high-latency networks, as the sender must wait for an ACK before proceeding. This contrasts with TCP,

which uses a sliding window mechanism to allow multiple packets to be "in flight" without immediate acknowledgments, thus utilizing the network more efficiently and increasing throughput. For example, in situations where packet loss occurs, this approach can significantly impact throughput. Since the sender must wait for an acknowledgment before sending more data, a lost packet (and its corresponding lost ACK) can halt the transmission until a timeout occurs and the packet is retransmitted (or in the assignment, the connection can stop immediately). TCP's use of SACK (Selective Acknowledgment) options and fast retransmission algorithms help mitigate this issue by allowing the sender to retransmit only the missing segments sooner, without waiting for their turn in a strict sequential order.

However, this simplification in du-proto makes the implementation much easier to understand and develop, as it removes the need for complex window management and congestion control algorithms. Since the protocol waits for an acknowledgment after each send, the sender and receiver are always synchronized in terms of state. There's less ambiguity about which packets have been received, making the protocol state easier to manage and reducing the complexity of handling out-of-order packets. Moreover, the sender does not need to buffer multiple packets while waiting for acknowledgments, as it only sends a new packet after receiving an ACK for the previous one. This reduces the memory requirements on both the sender and receiver. It trades off throughput and efficiency for simplicity and ease of understanding, which is suitable for the assignment purposes or simple applications.

**5. UDP vs. TCP From the Assignment**

**-** Socket Creation: A TCP socket is created using the socket() system call, similar to UDP. However, establishing a TCP connection requires the server to listen on a port and accept incoming connections. This is done using bind(), listen(), and accept() calls. The accept() call is blocking by default and waits for an incoming connection, returning a new socket specifically for this connection. For UDP, after creating a socket with socket(), you typically use bind() to associate the socket with a port (for servers). Unlike TCP, there's no need for listen() or accept() calls, as UDP is connectionless. Communication can start immediately using sendto() and recvfrom() calls, specifying the destination address for each packet.

- Connection Establishment: A TCP connection is established through a three-way handshake between the client and server, ensuring a reliable connection is in place before data transfer begins. This process is initiated by the client using connect(), which specifies the server's IP address and port number. UDP does not establish a connection. Instead, data packets (datagrams) are sent independently of each other. There's no handshake, making the protocol simpler but also connectionless and inherently unreliable. The lack of connection establishment reduces latency and overhead, beneficial for certain real-time applications.

- Data Transmission**:** Data transmission over TCP is stream-based. Data sent through a TCP socket is received in the same order, and there's no concept of message boundaries. TCP takes care of packet segmentation and reassembly, ensuring reliable and ordered delivery. Data transmission over UDP is message-based. Each call to sendto() sends one datagram, and each call to recvfrom() receives one datagram. Message boundaries are preserved, but delivery is not guaranteed, nor is the order of delivery.

- Reliability: TCP provides built-in mechanisms for reliability, including error detection, retransmission of lost packets, and in-order delivery. It also incorporates flow control (windowing / window sliding) and congestion control to adjust the rate of data transmission based on network conditions. UDP lacks built-in reliability and flow control mechanisms. Applications that require these features must implement them at the application layer.

Hence, TCP is used where reliability and data integrity are crucial. UDP might be chosen for applications where speed is more critical than reliability.