dpinit

Initializes a new dp_connection structure, setting default values, and returns a pointer.

Returns:

dp_connp - Pointer to the newly allocated and initialized dp_connection structure.

dpclose

Closes a dp_connection by freeing the allocated memory.

Parameters:

dpsession - Pointer to the dp_connection to be closed.

dpmaxdgram

Returns the maximum data payload size that can be sent in a single datagram.

Returns:

Int - The maximum datagram size in bytes.

dpServerInit

Initializes a server-side dp_connection, creates a UDP socket, binds it to the specified port, and prepares it for receiving data.

Parameters:

Port - The port number on which the server will listen.

Returns:

dp_connp - Pointer to the initialized dp_connection structure, or NULL on failure.

dpClientInit

Initializes a client-side dp_connection, creates a UDP socket, sets up the server address, and prepares it for sending data.

Parameters:

addr - The server's IP address as a null-terminated string.
Port - The port number on which the server is listening.

Returns:

dp_connp - Pointer to the initialized dp_connection structure, or NULL on failure.

dprecv

Receives data from a dp_connection, processes the received PDU, and copies the data payload into buff.

Parameters:

Dp - Pointer to the dp_connection.
Buff - Pointer to the buffer where the received data will be stored.
buff_sz - Size of the buffer.

Returns:

Int - The size of the received data payload or an error code.

dprecvdgram

Receives a datagram from the dp_connection, handles error checking, processes the PDU, and sends necessary acknowledgments.

Parameters:

Dp - Pointer to the dp_connection.
Buff - Pointer to the buffer where the received data will be stored.
buff_sz - Size of the buffer.

Returns:

Int - The number of bytes received or an error code.

dprecvraw

Receives raw data from the dp_connection socket and fills the buffer with the received data.

Parameters:

Dp - Pointer to the dp_connection.
Buff - Pointer to the buffer where the received data will be stored.
buff_sz - Size of the buffer.

Returns:

Int - The number of bytes received, or -1 on error.

dpsend

Sends data through the dp_connection, ensuring the data fits within the maximum datagram size.

Parameters:

Dp - Pointer to the dp_connection.
sbuff - Pointer to the data to be sent.
sbuff_sz - Size of the data to be sent.

Returns:

Int - The number of bytes sent or an error code.

dpsenddgram

Sends a datagram through the dp_connection, constructs the PDU with the data payload, and waits for acknowledgment.

Parameters:

Dp - Pointer to the dp_connection.
sbuff - Pointer to the data to be sent.
sbuff_sz - Size of the data to be sent.

Returns:

Int - The number of bytes sent or an error code.

dpsendraw

Sends raw data through the dp_connection socket.

Parameters:

Dp - Pointer to the dp_connection.
sbuff - Pointer to the data to be sent.
sbuff_sz - Size of the data to be sent.

Returns:

Int - The number of bytes sent, or -1 on error.

dplisten

On the server side, it waits for a connection request from a client and establishes the connection by exchanging necessary PDUs.

Parameters:

Dp - Pointer to the dp_connection.

Returns:

Int - Non-zero on success or an error code.

dpconnect

On the client side, the client initiates a connection to the server by sending a connection request PDU and waiting for acknowledgment.

Parameters:

Dp - Pointer to the dp_connection.

Returns:

Int - Non-zero on success or an error code.

dpdisconnect

Disconnects the dp_connection by sending a close request PDU and waits for acknowledgment.

Parameters:

Dp - Pointer to the dp_connection.

Returns:

Int - DP_CONNECTION_CLOSED on success or an error code.

dp_prepare_send

Prepares a buffer for sending by copying the PDU header into the buffer and returns a pointer to the position after the PDU header.

Parameters:

pdu_ptr - Pointer to the PDU to be sent.
Buff - Pointer to the buffer where the PDU will be copied.
buff_sz - Size of the buffer.

Returns:

Void* - Pointer to the buffer position after the PDU header, or NULL on error.

print_out_pdu

If debugging is enabled, print the details of an outgoing PDU.

Parameters:

PDU - Pointer to the dp_pdu to be printed.

print_in_pdu

If debugging is enabled, print the details of an incoming PDU.

Parameters:

PDU - Pointer to the dp_pdu to be printed.

print_pdu_details

Helper function to print the details of a PDU.

Parameters:

PDU - Pointer to the dp_pdu whose details will be printed.

pdu_msg_to_string

Converts the message type of a PDU to a human-readable string.

Parameters:

PDU - Pointer to the dp_pdu, whose message type will be converted.

Returns:

Char* - A string representing the message type.

dprand

The helper function for testing returns TRUE with a probability defined by the threshold; otherwise, it is FALSE.

Parameters:

Threshold - An integer between 1 and 99 representing the probability percentage.

Returns:

Int - 1 (TRUE) if the random number is less than or equal to the threshold, 0 (FALSE) otherwise.

Application Interface

`dpsend()`, `dprecv()`

Serve as the public API for sending and receiving data.
Provide a simple and user-friendly interface for the application layer.
Perform basic validation, such as checking if the buffer size is within acceptable limits.
Hide the complexity of the underlying protocol and transport mechanisms from the application.

Protocol Handling

`dpsenddgram()`, `dprecvdgram()`

Handle protocol-specific logic, such as constructing and parsing Protocol Data Units (PDUs).
Manage sequence numbers, acknowledgments, and error handling according to the protocol specifications.

Implement control messages and ensure reliable data transfer mechanisms (e.g., flow control, retransmissions).
Coordinate between the application interface and the raw data transmission layer.

Raw Data Transmission

`dpsendraw()`, `dprecvraw()`

Perform low-level socket operations to send and receive raw data over the network.
Abstract the details of the underlying transport mechanism (e.g., UDP sockets).
Ensure data is correctly transmitted and received at the byte level.
Provide a foundation upon which the protocol logic can be built.

Advantages

1. Separation of Concerns:

Each Layer has a clear and distinct responsibility, making the code modular.
Changes in one Layer (e.g., modifying the protocol logic) do not significantly impact the other layers.

2. Maintainability:

The modular design simplifies debugging and testing, as each Layer can be tested independently.
Enhancements or optimizations can be made to individual layers without affecting the overall system.

3. Reusability:

The bottom Layer can potentially be reused with different protocols, as it handles generic raw data transmission.
The top Layer remains consistent with the application, even if the underlying protocol changes.

4. Scalability:

New features or protocol versions can be added by extending the middle Layer without altering the application interface.

Considerations for Improvement

1. Error Handling Enhancements:

Implement more robust error handling and reporting mechanisms to provide more precise feedback to the application layer.

2. Thread Safety:

Consider adding synchronization mechanisms if the transport model is to be used in a multithreaded environment.

3. Abstraction Enhancements:

Define clear interfaces between layers to facilitate component substitution (e.g., swapping out UDP for TCP).

4. Protocol Flexibility:

Design the middle Layer to be more flexible to support multiple protocols or protocol versions.

Sequence numbers are fundamental to the protocol's design, ensuring reliable and ordered communication over an unreliable medium like UDP. Since UDP does not guarantee message delivery, order, or integrity, `du-proto` incorporates sequence numbers to manage these aspects at the application level. Sequence numbers help both the sender and receiver keep track of the order in which messages are sent and received. This is crucial for detecting missing, duplicate, or out-of-order messages. When a message requiring acknowledgment is sent, the sequence number ensures that the ACK corresponds to the correct message. This is important when multiple messages are in transit. If the receiver gets a message with an unexpected sequence number, it can identify an error, such as a lost or duplicate message, and take appropriate action. Sequence numbers assist in managing the data flow between sender and receiver, ensuring that both can handle the other with a manageable amount of data at a time. In `du-proto,` the sequence number (`seqNum`) is updated whenever a message that requires acknowledgment is sent or received. This update occurs during sending and receiving operations, ensuring synchronization between the communicating parties.

Sending Side (`dpsenddgram`):

```
//update sequence number after send
if (outPdu->dgram_sz == 0)
    dp->seqNum++;
else
    dp->seqNum += outPdu->dgram_sz;
```

If the datagram size (`dgram_sz`) is zero (indicating a control message like CONNECT or CLOSE), the sequence number is incremented by one. If actual data is being sent, the sequence number is incremented by the size of the data sent.

Receiving side (`dprecvdgram`):

```
//update sequence number after receive
if (err code == DP_NO_ERROR) {
    if (inPdu.dgram_sz == 0)
        dp->seqNum++;
    else
        dp->seqNum += inPdu.dgram_sz;
} else {
    //update seq number to ACK error
    dp->seqNum++;
}
```

Upon receiving a message without errors, the sequence number is updated similarly to the sending side. If an error is detected, the sequence number is still incremented to acknowledge the receipt of the erroneous message and maintain synchronization.

Reasons for Updating Sequence Numbers on Acknowledged Messages

1. Synchronization Between Sender and Receiver:

Updating sequence numbers ensures both parties have a consistent view of the communication session.

It allows the receiver to inform the sender about the progress of data reception.

2. Reliable Communication Over UDP:

Since UDP is unreliable, sequence numbers help implement reliability at the application layer.

They assist in detecting lost, duplicate, or out-of-order messages.

3. Matching Acknowledgments to Messages:

By including the sequence number in the ACK, the sender knows precisely which message the receiver acknowledges.

This is critical when multiple messages are in flight, preventing confusion between different messages.

4. Error Detection and Recovery:

If the sender does not receive an expected ACK with the correct sequence number, it can be deduced that the message was lost or corrupted.
This allows the sender to retransmit the message or take other error recovery actions.

5. Flow Control Management:

Sequence numbers can be used to manage the data transmission rate, ensuring that the receiver is not overwhelmed.
They help implement sliding window protocols or similar flow control mechanisms if needed.

The sender constructs a PDU (Protocol Data Unit) with the current sequence number.
After sending, it updates the sequence number based on the size of the data sent.
The receiver checks the sequence number of the incoming PDU.
If the sequence number matches the expected value, it processes the message and updates its sequence number accordingly.
It then sends an acknowledgment with the updated sequence number.
UDP is a connectionless protocol that does not guarantee:
- Delivery: Messages may be lost in transit.
- Order: Messages may arrive out of order.
- Integrity: Messages may be corrupted.

The protocol can detect and recover lost or corrupted messages by incorporating sequence numbers. Sequence numbers help reorder messages that arrive out of sequence, and discrepancies in sequence numbers can indicate data corruption.

Limitations of du-proto's Send-and-Wait Approach Compared to Traditional TCP

The du-proto protocol adopts a simple strategy where each send operation must be acknowledged before the next send is allowed. While this approach simplifies the protocol's design, it introduces several limitations compared to the TCP, which is widely used for reliable data transmission over networks.

1. Reduced Throughput and Efficiency

By requiring an ACK for each message before sending the next, du-proto effectively limits itself to having only one outstanding message in transit at any given time. TCP Utilizes a sliding window mechanism that allows multiple segments to be sent before receiving acknowledgments. This means TCP can have various packets in flight, maximizing available bandwidth. Du-proto's throughput is significantly reduced, especially in networks with high latency or bandwidth-delay products. The sender spends considerable time waiting for ACKs rather than transmitting data. TCP maintains high throughput by keeping the pipeline complete

with multiple unacknowledged packets and efficiently utilizes the network capacity. In Du-proto, the sender must wait for the ACK to traverse the long latency path before sending the following message, leading to severe underutilization of the link. However, TCP Can send many packets in succession without immediate ACKs, keeping the link busy and maximizing data transfer rates.

2. Increased Latency

Each message introduces a round-trip time (RTT) delay because the sender waits for an ACK before proceeding. This sequential operation increases the overall latency of data transmission. By not waiting for ACKs after every segment, TCP reduces the per-message latency, as multiple messages are in transit simultaneously. Applications requiring timely data delivery (e.g., video streaming) may experience delays and buffering issues with Du Proto's approach. Large file transfers take longer due to the cumulative effect of per-message RTT delays.

3. Inefficient Network Utilization

The sender's transmission capacity remains idle while waiting for ACKs, leading to poor utilization of the available bandwidth. TCP Continuously sends data up to the window size limit, ensuring the network is utilized effectively. The bandwidth-delay product represents the amount of data in transit in the network. Du-proto's send-and-wait approach cannot fill the network pipe, leading to suboptimal performance in high-speed networks.

4. Lack of Congestion and Flow Control Mechanisms

Without mechanisms to adjust the data transmission rate based on network conditions, du-proto cannot optimize performance or prevent congestion. However, TCP Implements congestion control algorithms (e.g., Slow Start, Congestion Avoidance) and flow control to manage the transmission rate adaptively. Du-proto may underutilize or, if modified to send more data, potentially overwhelm network resources without proper control mechanisms. TCP ensures fairness among multiple network flows, which du-proto does not address.

5. Inefficiency in Error Recovery

With only one message in flight, the Du-proto protocol cannot efficiently recover from packet loss by incurring significant delays. However, TCP Can detect lost packets through duplicate ACKs or timeouts and retransmit missing segments without halting the transmission process. Du-proto's performance degrades in networks with higher packet loss rates due to its simplistic error recovery mechanism. In contrast, TCP Maintains higher throughput by selectively retransmitting lost segments while sending new data.

Despite the limitations mentioned, the send-and-wait strategy simplifies the implementation of du-proto in several ways:

1. Simplified State Management
2. Simplified Error Handling and Retransmission Logic
3. No Need for Complex Flow and Congestion Control

4. Simplified Sequence Number Management
5. Reduced Memory Usage
6. Ease of Understanding and Maintenance

TCP Sockets

Setup:

Server Side:
1. `socket()`: Create a socket.
2. `bind()`: Bind the socket to an IP address and port.
3. `listen()`: Listen for incoming connections.
4. `accept()`: Accept a connection, creating a new socket for communication.

Client Side:
1. `socket()`: Create a socket.
2. `connect()`: Establish a connection to the server.

TCP maintains a persistent connection with state information and requires proper closing using `close()` or `shutdown()` to terminate the connection gracefully. `send()` and `recv()` are used to send and receive data over an established connection. Data is read as a continuous stream without inherent message boundaries. TCP ensures data is delivered reliably and in order. It handles retransmissions, acknowledgments, and error detection internally. Due to connection establishment and maintenance. Includes mechanisms to control data flow and manage network congestion. Applications requiring reliable communication, such as web servers, email clients, and file transfers. Data integrity and reliability without additional coding. Creates a new socket for each client connection via `accept().` Requires managing multiple sockets, often using multithreading or asynchronous I/O. Options like `SO_KEEPALIVE` can be set to detect dead connections. We may need to turn off using `TCP_NODELAY` for low-latency applications. Provides more detailed error reporting due to the connection state. Errors like connection reset or closed are reported, allowing the application to handle them appropriately. The protocol handles reliability and ordering. It also requires handling connection establishment and teardown, especially when dealing with multiple clients. Firewalls can track connections, making it easier to enforce security policies. Vulnerable to SYN flooding attacks due to the connection-oriented nature.

UDP Sockets

Setup:

Server and Client:
1. `socket()`: Create a socket.
2. `bind()`: Servers bind to a port; clients may bind if they need a specific port.

UDP does not require `listen()` or `accept()`; data can be sent and received without a formal connection. Each message (datagram) is independent. Since there is no connection, sockets can be closed directly when done. `send to ()` and `Recvfrom ()` is used for sending and receiving datagrams, including address information. This preserves message boundaries; each `sends ()` corresponds to a single `recvfrom().` It does not guarantee delivery, order, or error checking beyond basic checksums. Any required reliability or ordering must be implemented at the application layer. No connection setup reduces latency. It is faster but may result in packet loss if the network is congested. Real-time applications where speed is critical and occasional data loss is acceptable, like video streaming, gaming, or VoIP—reduced latency and overhead, suitable for broadcast and multicast transmissions. It uses a single socket to handle all incoming datagrams from multiple clients. Differentiates clients using address information in `recvfrom()` and `send to ().` Can enable broadcasting with `SO_BROADCAST.` Supports joining multicast groups using specific socket options. Errors are typically limited to immediate issues like invalid addresses or buffer sizes. Since there is no connection, errors related to the connection state do not occur, so there is no need to manage multiple connections. Applications must implement their mechanisms for reliability, ordering, and data integrity if needed. Firewalls may need to implement additional logic to handle UDP traffic. It is easier for attackers to spoof addresses since there is no connection handshake.
