Programming Assignment 4

Jared Chandavong

March 6, 2025

1 Function Descriptions

1.1 static dp_connp dpinit()

This function initializes a Drexel Protocol (DP) connection.

1.2 void dpclose(dp_connp dpsession)

This function takes in a Drexel Protocol (DP) session and closes it by freeing its allocated memory.

1.3 int dpmaxdgram()

This function returns the Drexel Protocol's (DP) maximum buffer size.

1.4 dp_connp dpServerInit(int port)

This function initializes a Drexel Protocol (DP) server over UDP by behaving as follows:

- Calls dpinit() to allocate a new dp connection. If initialization fails, it prints an error and returns NULL.
- Attempts to create a UDP socket. If socket creation fails, it prints an error and returns NULL.
- Fills up the sockaddr_in structure with appropriate information.
- Enables the server to reuse the port. If setting the option fails, it prints an error, closes the socket, and returns NULL.
- Bind the Socket. If binding fails, it prints an error, closes the socket, and returns NULL.
- Sets the dp initialization to true and returns the connection.

1.5 dp_connp dpClientInit(char *addr, int port)

This function initializes a Drexel Protocol (DP) client over UDP by behaving as follows:

- Calls dpinit() to allocate a new dp connection. If initialization fails, it prints an error and returns NULL.
- Attempts to create a UDP socket. If socket creation fails, it prints an error and returns NULL.
- Fills up the sockaddr_in structure with appropriate information.
- Copies the memory of the outbound address into the inbound address.
- Returns the connection.

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

This function recieves PDU of an inbound Drexel Protocol (DP) connection by behaving as follows:

Calls dprecvdgram(...) to store the datagram into a buffer and returns a received length. If the length is -16, then it returns a closed connection signal.

Convert the datagram into a DP PDU.

If the recieved length is greater than the size of the PDU header, then copy the memory of the payload into a buffer.

Return the size of the datagram stored in the inbound PDU.

1.7 static int dprecvdgram(dp_connp dp, void *buff, int buff_sz)

This function recieves a datagram of an inbound Drexel Protocol (DP) connection and validates its by behvaing as follows:

If the buffer size of the datagram is greater than the maximum DP datagram, then return an error that the DP buffer is oversized.

Calls dprecvraw(...) to get the number of bytes in the inbound datagram.

If the inbound bytes is less than the size of the PDU header, then an error code is made that the datagram is bad.

Copies the memory of the inbound datagram into a PDU struct. If the PDU's datagram size is greater than the buffer size, then an error code is made that the buffer is undersized.

Updates the sequence number and prepares the acknowledgement packet.

Creates and fills out an outbound PDU to be sent out.

If there is an error code, send a error message.

Handles different message types based off the inbound PDU. If an error occurs in any case, then return an error messeage.

Returns the number of bytes from the inbound datagram.

1.8 static int dprecvraw(dp_connp dp, void *buff, int buff_sz)

This function recieves raw UDP datagrams and validates them by behaving as follows:

If the inbound socket address is not initialized, then return an error.

Calls recvfrom(...) to store the datagram into a buffer.

If the received bytes is less than 0, then return an error.

Set the outbound socket address is set to initialized.
Prints the inbound PDU datagram and returns the number of bytes.

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

This function validates the buffer being sent by behaving as follows:
If the size of the send buffer is larger than the max DP datagram size, then return an error code that the DP buffer is undersized.
Calls dpsenddgram(...) to get the send size and returns it.

1.10 static int dpsenddgram(dp_connp dp, void *sbuff, int sbuff_sz)

This function sends a Drexel Protocol (DP) UDP datagram to a receiver by behaving as follows:
If the outbound address is not initialized, then return an error.
If the send buffer size is greater than the max buffer size, then return an error.
Builds the PDU and outbound buffer.
Copies the memory of the send buffer into the DP buffer.
Calls dpsendraw(...) to get the number of bytes in the outbound.
If the outbound bytes is not equal to the send size, then print a warning.
Updates the sequence number after the send.
Calls dprecvraw(...) to get an ack packet. If the inbound bytes is less then the size of the PDU header and not an ack packet, then print a expection error.
Returns the difference between the outbound bytes and PDU header size.

1.11 static int dpsendraw(dp_connp dp, void *sbuff, int sbuff_sz)

This function sends raw UDP datagrams and sends it to the appropriate receiver by behaving as follows:
If the outbound address is not initialized, then return an error.
Calls sendto(...) to store the bytes into the outbound address.
Calls print_out_pdu(...) to print the data in the outbound PDU.
Returns the number of outbound bytes.

1.12 int dplisten(dp_connp dp)

This functions ensures proper Drexel Protocol (DP) connection handshake between client and server by behaving as follows:

If the inbound socket address is not initialized, then return an error.
Calls dprecvraw(...) to receive a connection request. If the received size is not equal to the size of the PDU header, then return an error.
Calls dpsendraw(...) to send an acknowledgement packet. If the send size is not equal to size of the PDU header, then return a error.
Set the DP's connection to be true and returns true.

1.13 int dpconnect(dp_connp dp)

This function connects a client to a server by behaving as follows:

If the outbound socket address is not initialized, then return an error.
Calls dpsendraw(...) to send a connection request. If send size is not equal to the PDU header size, then return an error.
Calls dprecvraw(...) to receive an acknowledgement packet. If receive size is not equal to the PDU header size, then return an error.
If the recieved messeage type isn't CNTACT, then return an error.
Increments the sequence and set the connection to true.
Returns true.

1.14 int dpdisconnect(dp_connp dp)

This function disconnects a client from a server by behaving as follows:

Builds a close request.
Calls dpsendraw(...) to send a close request. If the send size is not equal to the PDU header size, then return an error.
Calls dprecvraw(...) to receive an acknowledgement packet. If receive size is not equal to the PDU header size, then return an error.
If the recieved messeage type isn't CNTACT, then return an error.
Calls dpclose(...) to close the DP connection, and returns closed code.

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

This function prepares a data buffer for sending a Drexel Protocol (DP) message by behaving as follows:

If the buffer size is less than the PDU header size, the return an error.
Sets the bytes in the buffer to 0.

Copies the memory of the PDU to be sent into the buffer.
Returns the sum of the buffer and the PDU header size.

1.16 void print_out_pdu(dp_pdu *pdu)

This is a helper for validating the printing of an outbound PDU. It takes a PDU and behaves as follows:

If the debug mode is not equal to 1, then return.
Prints out seperator for PDU details.
Calls print_pud_details(...) to print out the PDU's details.

1.17 void print_in_pdu(dp_pdu *pdu)

This is a helper for validating the printing of an inbound PDU. It takes a PDU and behaves as follows:

If the debug mode is not equal to 1, then return.
Prints out seperator for PDU details.
Calls print_pud_details(...) to print out the PDU's details.

1.18 static void print_pdu_details(dp_pdu *pdu)

This is a helper for printing out a PDU's details.

1.19 static char * pdu_msg_to_string(dp_pdu *pdu)

This is a helper that takes a PDU and returns the messeage type it has.

2 3 Sub-layers

2.1 Top-Layer

This layer is comprised of functions such as dpsend() and dprecv() in order to handle application-level data transfer and receive by preparing and sequencing data.

2.2 Middle-Layer

This layer is comprised of functions such as dpsenddgram() and dprecvdgram(). Here, data is encapsulated into PDUs and deals with the appropriate message types and acknowledgements.

2.3 Bottom-Layer

This layer is comprised of functions such as dpsendraw() and dprecvraw(). Raw UDP data is transfered of the network.

2.4 Is this good design?

I believe that this is good design, especially in terms of modulation. Since each layer is contained within their own functions handling specific responsibilities, it's much easier to error check and change accordingly. Furthermore, any modification to these functions will only cause isolated effects that can be used for testing without affecting the other layers.

3 Sequence Numbers

In the du-proto, sequence numbers are used to track the order of messages to be processed and to ensure that communication between client and server are consistent. As messages are sent to the receiver, ACK packets are returned with a sequence number for the sender to verify that they have received the correct message packet. It is also used to handle errors where missing sequence numbers can indicate packet loss.

We update the sequence number for ACK responses because since every message has its own sequence number, we want to avoid a possible issue of the sender ignoring the response for having a duplicate number. Furthermore, making sure that each message has a unique sequence number also allows for control messages

to be distinguish from one another so that every packet is processed correctly in order.

4 ACK vs TCP

I think that one major example of a limitation of this approach compared to traditional TCP is when we are transfering large amounts of data. The issue here is that transferring one packet at a time for an extensively large set of data, as well as waiting for each packet to receive an acknowledgement, will take a long time to process. In contrast, the traditional TCP approach would be able take on multiple packets at a time along with acknowledgement for each one, making it generally faster with reduced overhead.

5 UDP vs TCP

When we set up and managed UDP sockets, data is prepared for transfer as discrete packages, or datagrams, in an ordered sequence of specified designations before a connection between the sender and the receiver is established. In constrast, TCP prepares data as a continuous stream of back and forth responses after a connection between the sender and the receiver is established. Because of these differences, a key notion that can be derived is that UDP is faster in transferring data than TCP, but at a cost in that UDP is also less reliable in guaranteeing the delivery of packets unlike TCP.