Tristan West

1.

function: static dp_connp dpinit()

description: Allocates memory for and initializes the fields of a Drexel Protocol connection structure (dp_connection) to default values. Returns the struct, dpsession, after initialization.

function: void dpclose(dp_connp dpsession)

description: Frees the memory previously allocated for a provided Drexel Protocol structure passed as the argument dpsession. No return value.

function: int dpmaxdgram()

description: Returns the max buffer size to be used for a Drexel Protocol session.

function: dp_connp dpServerInit(int port)

description: Initializes a Drexel Protocol server session on the given port. This starts by attempting to create a default dp_connection struct, if this fails the function returns a NULL value. Next, it tries to get a socket number and server address from the dp_connection that was just created, and checks if the socket is valid. If not, the function returns with a NULL value. It then fills out the fields for the server address structure: family set to IPv4, internet address set to accept any, and port number set to the provided port parameter. It tries to set the reuse address socket option to true, returning NULL if this fails. Finally it will try to bind the socket to the server address, upon failure closing the socket and returning NULL. The function finishes by setting the is address initialized field and the value of the output socket address length field, and returning the Drexel Protocol connection structure.

function: dp_connp dpClientInit(char *addr, int port)

description: Initializes a Drexel Protocol client session on the given port, to connect to server addr. This function starts out the same as dpServerInit, diverging after creation of the socket file descriptor. When filling the server address structure, it sets the in address to the provided address parameter instead of accepting any address. It also initializes the out socket address, and copies its value from the in socket address. It ends by returning the filled out dp_connp structure.

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

description: Acts as a wrapper for dprecvdgram. It starts by receiving the data from the DP connection and storing it in the DP buffer. It checks if the length of the data received indicates that the connection has been closed, if so this value is returned and the function exits. If the connection wasn’t closed, it typecasts the

data from the DP buffer to a DP pdu. It checks if the length of data received exceeds the size of a DP pdu, and if so, copies the data from the DP buffer to a generic buffer. It ends by returning the size of of a dp_pdu datagram.

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

description: Receives a Drexel Protocol pdu over the DP connection dp into the provided buffer buff. Starts with a sanity check, if the provided buffer size exceeds the max datagram size, the function returns an error. It then gets the raw DP data and stores in a generic buffer. It checks if the number of bytes read is less than size of a DP pdu, and sets the error code accordingly. It then copies the data from the generic buffer to a pdu buffer, and once sets the error code if there is a size mismatch. It then checks if there have been any error codes, and uses this to adjust the sequence number. It then creates a new DP pdu and sets its fields, with the mtype field requiring special treatment to check for errors. The function finishes by returning the number of bytes read in.

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

description: Reads in raw data over DP connection dp into the provided buffer buff. Starts by checking if the socket has been initialized, if not returns with an error. Next tries to recv from the UDP socket provided by the dp struct. If this fails, the function returns with an error. It then sets the out socket initialized flag true, creates a new in pdu with the data held in the buffer, prints this pdu, and returns the number of bytes received.

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

description: Acts as wrapper for dpsenddgram. Starts by checking if the provided buffer size exceeds the max size of a dp datagram, and returns an error if so. It then tries to send the dp datagram, returning the size of the the datagram sent.

function: static int dpsenddgram(dp_connp dp, void *sbuff, int sbuff_size)

description: Sends a DP datagram over DP connection dp from send buffer sbuff. Starts by checking if the out socket address of dp is initialized, if not it returns with an error. It then checks if the send buffer size exceeds the max DP buffer size, if so returns with an error. It then creates a new dp pdu from the dp buffer, and initializes its member values. It then copies the data from the send buffer to the dp buffer, and calculates the total number of bytes being sent. It then sends the raw bytes from the dp buffer, and compares the number of bytes send to the expected amount, and prints a warning if they do not match. It then updates the sequence number, and tries to receive the acknowledgment back from the receiver, printing a warning if there is a mismatch in mtype. It returns the number of bytes sent.

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

description: Sends the raw bytes over DP connection dp from send buffer sbuff. Starts by checking if the dp out socket is initialized, returning an error if not. It then creates a new out pdu from the send buffer, sends the raw bytes over the udp socket provided by dp, and prints out the pdu. It returns the number of bytes sent.

function: int dplisten(dp_connp dp)

description: Listens for an incoming connection over DP connection dp. Starts by checking if dp socket is initialized, returns error if not. Creates a new empty dp pdu, and tries to receive raw bytes into the new pdu. Checks the number of bytes received, if the number is unexpected return an error. Sets the pdu's mype to ack, and increments the sequence number for the dp connection and pdu. It then sends back the acknowledgment, once again checking the number of bytes. If successful, sets is connected field true and returns true.

function: int dpconnect(dp_connp dp)

description: Connects to DP connection dp. Starts by checking if out socket is initialized, if not returns an error. Creates a new pdu, sets mtype to connect, and the sequence number to dp's seqnum, and datagram size to 0. Attempts to send the pdu to start the connection, then attempts to receive the acknowledgment back. If there is a mismatch between the expected and actual amount of bytes received or sent, or the mtype, returns an error. Increments dp sequence number, sets dp is connected to true, prints a confirmation message, and returns true.

function: int dpdisconnect(dp_connp dp)

description: Disconnects from DP connection dp. Starts by initializing a dp pdu, using protocol version drexel protocol, mtype close message, sequence number set to dp's sequence number, and datagram size 0. Attempts to send the close request over dp, and then receive the acknowledgment back. If there is a mismatch between expected and actual bytes recieved/sent or mtype, return an error. Returns connection closed (-16).

2.

The outer layer functions (dprecv, dpsend) are responsible for performing sanity checks before calling the underlying middle layer function to ensure they will succeed. The middle layer functions (dprecvdgram, dpsenddgram) build out their respective pdus using the raw data acquired by calling their respective inner layer functions, as well as performing robust error checking. The inner layer functions (dprecvraw, rpsendraw) are responsible for making the direct calls to the socket library to send and receive the raw bytes to be passed up the chain. I like separation of responsibility for the sake of abstraction for the end user, however I think the middle layer datagram functions are doing too much of the heavy lifting. I think readability could be significantly improved if much of the

error checking could be moved into the outer layer functions, and the middle layer functions focus more closely on formatting the pdus.

3.

The sequence number of the Drexel Protocol connection is used to keep track of the order in which each datagram is sent between the host and client. When looking for an acknowledgment, it is expected that it will be the very next thing received, so the correct ack will be seq + 1.

4.

With traditional TCP, you would be able to send new data along with an ack in a response, so with this arrangement there will be more send and recv calls to achieve the same effect since we are always waiting for ack. However this simplified the implementation as we never have to worry about handling the scenario where a response included additional along with the ack.

5.

With UDP sockets, the programmer is left to do much of the work in managing the order of data segments themselves, where as TCP does the work of ensuring data is ordered correctly. With TCP, after a socket connection is open it is maintained until the transaction is complete, where as in UDP the socket connection only lasts to send or receive that datagram. UDP also requires more work from the programmer in formatting these datagrams for transmission.