CS472 Written Assignment 1

Benjamin Boardman

August 2025

1 Go through the code in `du-proto.c` and make sure you understand it. For each function, come up with the most appropriate description/comment block and include the function name and description.

1. `static dp_connp dpinit()`
   This function initializes a dp_connp pointer, which stores connection information. It returns a default version of the connection struct.

2. `void dpclose(dp_connp dpsession()`
   This function takes and then frees a dp_connp pointer, freeing the associated memory.

3. `int dpmaxdgram()`
   Thus function returns the maximum size for a single datagram, by default 512 bytes.

4. `dp_connp dpServerInit(int port)`
   This function does all of the setup necessary to initialize a du protocol server listening on the given port. It calls `dpinit()` to generate a `dp_connp` data structure, opens and configures a socket, and binds to a server, returning the filled-out connection struct.

5. `dp_connp dpClientInit(char *addr, int port)`
   `dpClientInit` does the corresponding setup for a du-proto client. It sets up the `dp_connp`, opens a socket, sets a server address based on arguments, and then returns the connection struct. Since this is a connectionless UDP-style protocol, it does not actually handle the connection portion of this set up.

6. `int dprecv(dp_connp dp, void *buff, int buff_sz)`
   `dprecv()` is largely a wrapper for `dprecvdgram()`. It passes its parameters along to the above function, copies the datagram received to the argument `buff`, and then returns the size of the datagram.

7. `static int dprecvdgram(dp_connp dp, void *buff, int buff_sz)`
   `dprecvdgram()` starts by calling `dprecvraw()` to receive `buff_sz` bytes of data into `buff`. If there is not a full du proto PDU, it throws an error; otherwise, it sets up a PDU. If the datagram in the PDU is larger than the buffer, it throws another error. If there is no error, it updates the sequence number with the size of the datagram; otherwise, it increments the sequence number. It then prepares another PDU with no datagram to ACK the received message. If an error has occurred, it will ACK an error message back to the sender; otherwise, it will send an ACK based on the sent message. If the sent message was a "connection closed" message, it will close the connection after ACKing the message. Finally, it will return the error status, or zero if there was no error. All ACK messages are sent using `dpsendraw()`.

8. `static int dprecvraw(dp_connp dp, void *buff, int buff_sz)`
   This function reads `buff_sz` bytes from the socket stored in `dp`, detects the PDU, prints it to the console, and then returns the number of bytes read. There's an optional (currently disabled) bit of debug code that also detects and prints the payload.

9. `static int dpsend(dp_connp dp, void *sbuff, int sbuff_sz)`
   Like `dprecv()`, this function does very little of the work of sending data. It makes sure the buffer is large enough to handle a datagram and thencalls `dpsenddgram()`, returning the send size that function returns.

10. `static int dpsendgram(dp_connp dp, void *sbuff, int sbuff_sz)`
    `dpsenddgram()` handles the bulk of the overhead related to a du protocol send. It calls `dpsendraw()` to actually send the data, updates the sequence number, and waits for an ACK, which comes in the form of a call to `dprevcraw()`. If the ACK is of a different type then a SND ACK (acknowledgement of data sent), an error message is printed. The size of the sent datagram is then returned.

11. `static int dprsendraw(dp_connp dp, void *sbuff, int sbuff_sz)`
    This function fills a similar role to `dprecvraw()` in that it actually does the data transfer. `dpsendraw()` sends `sbuff_sz` byes from `sbuff` to the `udp_sock` in `dp`, prints the PDU, and then returns the number of bytes sent.

12. `int dplisten(dp_connp dp)`
    `dplisten()` is intended to be called by a server waiting for a client to connect. It waits for a PDU to be received using `dprecvraw`, ACKs it using the `CNTACK` status, increments the sequence number, sets the `isConnected` field of `dp` to true, and then returns true.

13. `int dpconnect(dp_connp dp)`
    This is the client-side companion function to `dplisten()` above. It sends a PDU with type `CONNECT` to the server, waits for an ACK of type `CNTACK`,

increments the sequence number, sets the `isConnected` field of `dp` to true, and then returns true.

14. `int dpdisconnect(dp_connp dp)`
    Here is how a connection is closed. This function sends a PDU with type `CLOSE`, waits for an ACK of type `CLOSEACK`, and calls `dpclose()` to end the connection.

15. `void * dp_prepare_send(dp_pdu *pdu_ptr, void *buff, int buff_sz)`
    This simple function attaches the `dp_pdu pdu_ptr` to the buffer `buff` and returns the address of the start of the payload in the buffer.

2 What are the specific responsibilities of the sublayers? What are their responsibilities? Is this a good design?

The three sublayers here function concentrically, which is not how I would have imagined a sublayer design. Specifically, there is the "raw" layer composed of the two `_raw` functions, inside the "dgram" layer consisting of the two `_dgram` functions, which lives inside the send/receive layer itself.

In the raw layer, the functions are the ones actually send and receive data. They do very little outside of calling the `sendto()` and `recvfrom()` and some very basic error checking. Surrounding the raw layer is the dgram layer. This layer makes sure the datagram size isn't exceeded, processes the PDU attached to the data, and handles the sending/receiving of acknowledgements between server and client. The bulk of the process (maybe 60%) is contained in this layer. The outer layer of `dpsend()` and `dprecv()` does very little - they effectively just call the dgram sublayer and close the connection if necessary.

Is this a good design? I believe it could be better, but it is also somewhat necessary. Certainly the send/recv layer and the dgram layer don't need to be separate, but the dgram layer actually calls into the raw layer multiple times for sending and receiving ACK messages, which provides useful code reuse.

3 How does this protocol use sequence numbers? Why is the sequence number updated for things that need to be acknowledged?

This protocol uses sequence numbers to track the total number of bytes sent between the sender and recipient. The sequence number is updated for acknowledgments and errors to indicate that communication has occurred, even if no data was sent during the transaction. Alternatively, even the null pointer contains a single byte, so technically a data byte is sent.

4 The DU proto requires an ACK after every send rather then after a group of sends. What is a potential limitation of this? How did this simplify the DU proto?

The biggest limitation is that sending many small datagrams takes significantly longer. If several small datagrams are sent in quick succession, each one will need to be ACKed before the next one can be sent, effectively doubling the time to complete the whole transaction. However, this represents a massive simplification of the protocol since a linear and always-identical receive and send procedure can be followed. On the receive side, it goes receive-process-ack, while the sender does send-wait-receiveACK before sending new data. Presumably, a more complex system would have the receiver having some sort of separate process that waits for a specified period of time before ACKing , while the sender would give an error if a certain amount of time passes after a send with no ACK.

5 Briefly describe some of the differences associated with setting up and managing UDP sockets as compared to TCP sockets.

The first difference I notice is that the client side doesn't need to call the POSIX function `connect` and the server doesn't need to use the POSIX `listen` or `accept`. Instead, simply sending a DU Proto PDU with a connect message and having it acknowledged is enough for the client, and all the server cares about is whether it has received that PDU. This fits with UDP being a connection-less protocol without any sort of handshake, beyond the "I exist" purpose of this send/receive. The use of `sendto()` and `recvfrom()` containing the `sockaddr` as a parameter is used to support this, since it isn't connected to actively so it must be sent directly. However, the man pages are not clear on whether these are specifically for UDP-style use, so this is more of an inference.