CS472: Introduction to Network Programming
Homework 4

Ayo Adetunji

Question 1: 1: Comments for du-proto.c

- **dpinit():**

  Parameters: None

  Returns => a pointer "**dpsession**" which is a pointer to a dp_connection structure.

  This function is used to create and initialize a dp_connection structure. The values of all the members of the struct are set to the default values

- **dpclose():**

  Parameters => dpsession: dpconnp

  Returns => None
  This function takes in a pointer to a dp_connection struct and releases the memory allocated

- **dpmaxdgram():**

  Parameters=> None

  Returns => the DP_MAX_BUFF_SZ macro
  This function simply returns the maximum size of a datagram for the drexel protocol

- **dpServerInit():**

  Parameters => port: int

  Returns => dpc: dp_connp

  This function creates the structure "dpc" of type dp_connp using dpinit(). A socket is created to allow communication using the UDP protocol(SOCK_DGRAM) and the udp_sock member is updated as needed. The function then updates the members of the struct starting with the information needed to set up the connection, including the address type IPv4, the port passed as the sole parameter, and utilizing the INADDR_ANY macro, the listen address to be used will be 0.0.0.0, which will listen on any available interface on the "server". The socket options on the udp_sock are updated to allow the reuse of ports with wait time removed. Then utilizing the IP+port information set previously, the socket is bound to the IP address and port set using the bind() function to set the IP and port to be used for communication over that socket.

- **dpServerInit():**

  Parameters => port: int

  Returns => dpc: dp_connp

  This function creates the structure "dpc" of type dp_connp using dpinit(). A socket is created to allow communication using the UDP protocol(SOCK_DGRAM) and the udp_sock member is updated as needed. The function then updates the members of the struct starting with the information needed to set up the connection, including the address type IPv4, the port passed as the sole parameter, and utilizing the INADDR_ANY macro, the listen address to be used will be **0.0.0.0**, which will listen on any available interface on the "server". The socket options on the

udp_sock are updated to allow the reuse of ports with wait time removed. Then utilizing the IP+port information set previously, the socket is bound to the IP address and port set using the bind() function to set the IP and port to be used for communication over that socket. This function returns the dp_connection struct that contains information for a server configuration and all that is needed to listen and receive a connection from a client

- **dpClientInit():**

  Parameters => addr: char; port: int

  Returns => dpc: dp_connp

  This function sets up all the structures needed to allow a client to connect to a remote server using the addr and the port provided as parameters. Includes the same process of creating a socket utilizing udp too (SOCK_DGRAM) and IPv4. The function also returns a dp_connp struct with all the information needed for a client to connect to the destination server, including port, destaddr, and destport.

- **dprecv():**

  Parameters => dp: dp_connp; buff: void; buff_sz: int

  Returns => an integer equal to the size of the datagram received

  This function calls the function to receive the data sent over the connection setup in the dp_connection struct passed a parameter into the function, and all the received data is written to the local "_dpBuffer". When the connection closed indicator is returned by the dprecvdgram() function which is used to do the receives, the function returns that value. Otherwise, _dpBuffer is typecasted to "dp_pdu" to fit the dp_pdu structure defined. A check that the size of the data received is not just the Drexel Protocol PDU and then the data received is copied into the buffer (buff) provided as part of the parameters. The size of the datagram received is then returned.

- **dprecvdgram():**

  Parameters => dp: dp_connp; buff: void; buff_sz: int

  Returns => an integer equal to the total bytes received.

  This function receives the raw data from the socket utilizing the dprecvraw() function. Checks that the number of bytes received is greater than the size of the Drexel Protocol PDU size, then fits the received data into a struct that defines the protocol and confirms that the bytes received are not larger than the size of the buffer provided. If no errors are hit, then checks are completed to update the sequence number for the ack messages. The function then creates another struct for the acknowledgment with all the required information, including the sequence number if there are no errors. Finally updates the message type for the ACK and utilizes the dpsendraw() function to send the acknowledgement. Finally returns the size of the bytes received.

- **dprecvraw():**

  Parameters => dp: dp_connp; buff: void; buff_sz: int

  Returns => an integer equal to the total bytes received.

  This function receives all the data from the socket and addressing information configured in the "dp" parameter, stores said data in the buffer(buff) passed in. Usually the MSG_WAITALL flag would block until the request is satisfied, but this is only when the socket is configured with "SOCK_STREAM", but is said to have no effect with "SOCK_DGRAM" configured sockets. This function simply receives the data, while noting the expected source host. The recvfrom() is used here as with udp, a connection is not formed using the connect() function as in TCP, so explicitly setting the host/remote location to receive packets from is needed

- **dpsend():**

  Parameters => dp: dp_connp; sbuff: void; sbuff_sz: int

  Returns => an integer equal to the total bytes sent.

  The function restricts the size of the buffer to be sent and calls the dpsenddgram() function to send the data in the buffer if the sizing is within the limits.

- **dpsenddgram():**

  Parameters => dp: dp_connp; sbuff: void; sbuff_sz: int

  Returns => an integer equal to the total bytes sent minus the size of the Drexel Protocol PDU

  This function builds out the PDU for the datagram to be sent, takes the contents in the send buffer sbuff and merges both in the local buffer _dpBuffer then utilizes the dpsendraw() function to send the header and the additional data to be sent. The function then updates the sequence number and then waits for an acknowledgment from the destination host.

- **dpsendraw():**

  Parameters => dp: dp_connp; sbuff: void; sbuff_sz: int

  Returns => an integer equal to the total bytes sent.

  This function sends data over the socket configured in the dp parameter, and sets explicitly the destination where the data should be sent to, and returns the number of bytes sent. The sendto() is used here as with udp, a connection is not formed using the connect() as in TCP, so explicitly setting the destination for each packet is needed.

- **dplisten():**

  Parameters => dp: dp_connp;

  Returns => True/False 1/0 or an integer

  This function is useful when setting up a server and listens for a connection from a potential client on the socket defined in the dp struct passed into the function. The function expects the first thing that happens to be a packet requesting a connection to be formed. If that passes, the function then uses the same packet to update the message type, and sequence number and then sends an acknowledgment back, and updates the dp_connection struct (dp) to show that a connection has been successfully created with the client.

- **dpconnect():**

  Parameters => dp: dp_connp;

  Returns => True/False 1/0 or an integer

  This is used on the client to create a connection to the server and send the correct control packets to create a connection for the Drexel Protocol. The function sets up the connect packet, sends that, and then waits to receive an acknowledgment, increments the sequence number by 1 given that only control data is sent, and updates the state of the dp_connection struct to isConnected to being true and returns. has several checks on the received and sent data and the sizes. Returns the value of the DP_CONNECTION_CLOSED macro to signify that the server acknowledged the closure of the connection.

- **dpdisconnect():**

  Parameters => dp: dp_connp;

  Returns => integer; Success: -16/DP_CONNECTION_CLOSED

  This function sets up the control packet to be sent to close a connection with the server. Sends the packet and waits for an acknowledgment from the server and then uses dpclose() to free all the memory allocated for the connection (dp) information like socket data and IP address is cleared.

- **dp_prepare_send():**

  Parameters => pdu_ptr: dp_pdu; buff: void;buff_sz: int

  Returns: None/NULL

  This function, although seemingly not used anywhere else takes in the Drexel Protocol PDU, a buffer, and the size of the buffer, which confirms the buffer can at least contain the PDU. The function empties the buffer, buff, and then copies the PDU struct found at the memory location the pointer pdu_ptr points to.

- **print_out_pdu():**

  Parameters => pdu: dp_pdu

  Returns: None/NULL

  This function simply is a helper to print out the details contained in a Drexel Protocol PDU being sent out.

- **print_in_pdu():**

  Parameters => pdu: dp_pdu

  Returns: None/NULL

  This function simply is a helper to print out the details contained in a Drexel Protocol PDU received.

- **print_pdu_details():**

  Parameters => pdu: dp_pdu

  Returns: None/NULL

  This function prints out the contents of the Drexel Protocol PDU to stdout.

- **pdu_msg_to_string():**

  Parameters => pdu: dp_pdu

  Returns: message type: char */string

  This function returns the STRING translation of the message value in the Drexel Protocol PDU received or sent.

- **dprand():**

  Parameters => threshold: int

  Returns: integer

  This function is a helper that is aimed at generating random errors from time to time to test resilience. The code takes a threshold value, and using that along with the pseudo-random number generator functions srand() to seed by utilizing the number of seconds since epoch as the seed value to allow reproducibility. Then utilizing the result of the rand() function which is dependent on the seed value set previously, modulo with 100 plus 1, the function compares with the threshold and returns based on the result of the comparison.

Question 2: Sublayers

- The base dpsend/dprecv functions are the first layers that do minimal work and make the calls to helpers that do the main work. But both of these do some checks around the data to be received or sent, one to make sure that the data to be sent is within the limits set and the other makes sure that the non-header data received is moved into the correct buffer provided by the caller to read from.

- The *dgram functions do a lot of heavy lifting including sending/receiving acknowledgments and checking for various errors like the size of the total received bytes being smaller than the size of the header which indicates a clear error. These functions do a lot of the control around the sending and receiving of the packets and what errors should be returned and sent as a reply to the sender when needed.
- The *raw() functions aim at doing the actual recvs and sends and dealing with the errors specific to that and returning said errors as needed. This is where the bytes are sent out and received in.
- I think the structure works, most especially the *dgram() and *raw() functions, and I would argue that the contents of the dpsend() and dprecv() functions could exist within the *dgram functions, but is useful. But separating the sending of the raw bytes and handling of the contents that are related to the protocol in another, seems to make debugging a bit easier, but also somewhat make the code a bit harder to read.

Question 3: Sequence Numbers

- In the DU protocol the sequence numbers are updated/incremented in the following situations:
  - A control message is received successfully
  - There was an error that was received and needed to be acknowledged
  - A certain number of bytes of non-header data has been sent and received
- The sequence numbers are also utilized as part of the dp_connection struct to allow for easier management of the value as long as the connection exists.
- The sequence numbers are used to keep track of how much data has been received/sent
- The need to increment the sequence number when ACKing is needed to signify to the sender that the correct number of bytes were received by the receiver and said increment can also be used to set the starting point for the next datagram to be sent. The increment is useful as a means to separate the received data as needed.

Question 4: ACKd

- One limitation of this approach is that something like TCP Piggybacking or a Cumulative Acknowledgement is not possible as the sender blocks until it has confirmed that the receiver has gotten the data. This also means that there is significantly more overhead with this protocol's communication. This also limits the sender to sending a single packet every time as it has to wait for the receiver to always reply, hence possibly not fully utilizing the bandwidth. There are also increased chances of stalling in this protocol as there seems to be no clear resend mechanism, and in the situation where the sender/receiver is unable to reply, the implementation will likely continue blocking for an unknown amount of time.
- This is much simpler to implement compared to TCP because the features that exist in TCP such as the window-sizing mechanism and timeout mechanisms are not implemented in the DU protocol. The DU-Protocol simply sends waits to hear back, and doesn't seem to set a time on how long to wait, leading to possible unexpected situations, but it does implement the core ability to confirm that a packet was received as expected and of the expected size and also in the correct order. If we were to send several at a time, we would likely need to rearrange all received datagrams ourselves, but this is an area the DU protocol ignores for the simpler implementation of enforcing all packets be sent in order and not continuing without that confirmation

Question 5: UDP vs TCP

UDP Socket	TCP Socket
Uses a SOCK_DGRAM (datagram) based socket	Uses SOCK_STREAM (stream) based socket
Does not require calling the "connect()" function on the client side to send but utilizes the recvfrom and sendto functions to sed the host to receive from and set where to send specifically	Requires the connect() function on the client side in order to continually send without using sendto or recvfrom and rather use the recv/send functions
On the server setup, only requires the socket creation, and binding to a port	Requires socket creation,binding to a port, and then listening on said port, and then for each connection being ready to accept using the accept() the connections

- Generally the TCP sockets require a lot more steps as there is a need to maintain a connection between the hosts, for the exchange of data across the applications(binaries in our case so far). But with UDP, given that it is connectionless, each packet simple needs to know where to go when needed and thereby the setup is minimal for both server-side and client-side configurations.