Ethan Gibson
CS-472-HW5 UDP

<u>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 in the written.pdf file.</u>

ANSWER: <u>SEE BELOW</u>

<u>2. I used a 3 sub-layers for various parts of transport model. If you look at dpsend() and dprecv(), each one of these is supported by two additional helper functions. For example dpsend() with dpsenddgram() and dpsendraw(). The same model is used by dprecv(). What are the specific responsibilities of these layers? Do you think this is a good design? If so, why. If not, how can it be improved?</u>

<u>ANSWER:</u>

For ease of identification, I will refer to the dpsend() and dprecv() as layer 1, dpsenddgram() and dprecvdgram() as layer 2, and dpsendraw() and dprecvraw() as layer 3.

With layer 1, this layer is what is called to send payloads. It is the function that calls layer 2, which calls layer 3. It is the helper function to send, making it easy to just simply call the send function. It makes it easy for the application to easily just call a send function, and let the backend handle the rest.

With layer 2, this layer is responsible for casting the bytes to the drexel PDU. This function fills out the PDU fields, and adds the payload to the memory following the header. It is also responsible for calling the layer 3 functions.

Layer 3 is responsible for actually performing the send or recv functionality, using bytes. It ensures that all data is properly sent/recv'ed.

I do think this is a good design, it helps separate out responsibility, making the code easier to understand and follow along, also helps with error checking. One recommendation I have to improve on this model is to also utilize the sequence numbers to check for an acknowledgement. In doing so, it would allow us to add in handling for lost or missing packets.

<u>3. Describe how sequence numbers are used in the du-proto? Why do you think we update the sequence number for things that must be acknowledged (aka ACK response)?</u>

<u>ANSWER:</u>

Sequence numbers are used to tell the recv'er and sender how many bytes have been sent between one another, and that we are ready for the next bytes after n. The way it is set up in this du-proto is that whenever an ack is sent, the seqnum is increased by just 1; however, if a payload is sent, then the seqnum is set to the length of the payload plus the length of the previous data recv'ed. The purpose of updating the sequence number for things that need to be acknowledged allow, while not implemented in this, to check for duplicate/lost/out of order messages.

4. To keep things as simple as possible, the du-proto protocol requires that every send be ACKd before the next send is allowed. Can you think of at least one example of a limitation of this approach vs traditional TCP? Any insight into how this also simplified the implementation fo the du-proto protocol?

ANSWER:

The biggest limitation is that this doesn't allow for multiple messages to be sent at once, making it significantly slower than traditional TCP, since only one message can be handled at a time. At the same time, this also simplified the implementation for the du-proto protocol. For example, it allows us to simplify the acknowledgement process, since all we have to do is either increase the seqnum by 1 for control messages, or keep a running track of the number of bytes recv'ed and update the seqnum to that. Since we are only handling one message at a time, if an ack is not recv'ed the sender/receiver are both referencing the same message, so it can easily be resent if needed, though it is not done in this assignment. If this were traditional TCP, we would have to implement things like the "sliding window" functionality and timers to keep track of out-of-order/lost messages.

5. We looked at how to program with TCP sockets all term. This is the first example of the UDP programming interface we looked at. Briefly describe some of the differences associated with setting up and managing UDP sockets as compared to TCP sockets.

ANSWER:

Setting up and managing TCP vs UDP sockets is quite different, where TCP requires much more setup to get an active connection between a client and server. All that is required for UDP is setting up the server by creating a socket, and binding to that socket. For the client with UDP, a socket just needs to be created, and after that it is able to just send and recv using the sendto and recvfrom functions. For TCP, we are required to set up sockets, use listen(), bind(), and accept() for the server, and the client has to create the socket and then call the connect() function. Additionally, with UDP the developer has to handle things like acknowledgement handling and creating a sort of "reliability" like we did with the seqNum in this protocol, with TCP, that is already built in.

dpinit()
This function first allocates memory for a new dpsession. Following that, it zeros
out the dpsession struct so all fields are 0, using bzero.

Following that, it sets all init values to false, like for in/out socket address, and then sets the size of both addresses to the size of the sockaddr_in struct.

It sets the sequence number to 0, since nothing has been sent/recv'ed yet, makes the boolean value `isConnected` to false,
and also puts the session in debug mode.

Finally, it returns a pointer to the connection structure.

dpclose()
This function just simply frees the dynamically allocated memory for a dp_connection performed in dpinit().
It does not close any sockets, and just simply cleans up the memory with free.

dpmaxdgram()
This function simply returns the max payload size we can use.

dpServerInit(int port)
This function intializes a new server utilizing the drexel protocol, using the port specified in the parameter.

It calls dpinit() to intialize a new connection, and checks to ensure it was created properly, otherwise, it returns a NULL pointer. If it was created successfully, the sock variable is set to the address of the udp_sock field in the struct, likewise with the server address.

Then a socket file descriptor is created with the socket() function and stored in the udp_field in the struct. This creation is checked and if it failed returns NULL.

The connection struct is then populated with the server address and port (which is converted to network-byte order).

At this point, we have a valid socket, and the function continues to configure it with `setsockopt`, and we use SO_REUSEADDR so that the port is reused immediately, allowing the program to not have to wait until the port is released by the OS. If this fails, we return NULL.

Next, we bind the socket, and once again check for a successful bind, if there is an error, the function returns NULL

If the program gets past this point, it means that we have a valid socket and no errors have arised, so we can say that the in address has been initialized (isAddrInit = true), and we can set the out address length.

We then return the initalized dp_connection struct.

In short, this function initalizes a valid server socket

dpClientInit(char *addr, int port)
This function is very similar to the server init function listed above, as it initalizes a valid client socket, and returns the connection struct (where the socket fd is stored).

This function calls dpinit() making a new connection struct for the client, and returns NULL on failure. A udp socket is created using the socket() function, again returning NULL on failure.

If successful, we set the address using `inet_addr` which translates an string IP address to bytes, the port with htons(), which converts it to network-byte order. We also set the boolean value stating that the out address has been initialized, and set the length of the out address to the size of the sockaddr_in struct.

A main difference between the server init function and this is what comes on the next line. The function copies the outbound address to the inbound address using `memcpy()`. This works because the client and server address can be treated as the same in the context of the client.

Finally, the function returns the valid dp_connection struct for the client.

dprecv(dp_connp dp, void *buff, int buff_sz)
This function takes a valid dp_connection struct, buffer, and integer representing the buffer size as parameters. It returns the size of the recv'ed datagram.

This function utilizes the dprecvdgram() function defined below, which will be explained in more detail then.

Overall, this function recv's a PDU into a buffer, type casts it to a dp_pdu struct, and returns the length.

dprevcdgram(dp_connp dp, void buff, int buff_sz)

This function is responsbile for recv'ing a full PDU. It calls the dprecvraw() function, which will be discussed further on, but utlimately it returns the number of bytes read in.

This number is then checked to ensure we recv'ed the entire pdu, and if not, returns an error.

Following that, the buffer holding the recv'ed pdu is copied to a PDU structure and ensured it is the correct size.

At this point, we have a recv'ed PDU in a PDU structure. The function then updates the sequence number. For example, if there is no payload, then the seq number is just increased by 1, signifying an ack response. Otherwise, if there is a payload then the sequence number is increased to the pdu size it recv'ed. If there was an error, than the sequence number is increased by 1 to acknowledge the error.

From there, we form the outbound PDU. The Protcol, dgram size, sequence number, and error code are set. The sequence number and error codes are based off of the recv'd PDU results.

If there was no error, then there is a switch case on the message type. If the message type is of type send, then either the message or acknowledgement is sent. if the message type is of type close, then the function sends a close acknowledgement and closes the connection. For the sends, the function calls dpsendraw(), which will be discussed later. For all of the sends, we are checking to ensure that the entire PDU was sent, and if not, raise an error.

Finally, the function returns the number of bytes that were recv'ed in.

dprecvraw(dp_connp dp, void buff, int buff_sz)
This function is responsible for the actual recv operation, and returns the number of bytes recv'ed.

It first checks to ensure that the connection was setup properly, by looking at the isAddrInit, field, which will be true or false. If false, it returns an error.

Otherwise, it uses recvfrom() to recv the UDP socket into a buffer, and uses MSG_WAITALL to ensure all data is full recv'ed before continuing. It then checks the recv for errors, and if one occured, returns -1.

If we get past this point, it means we had a valid recv and that the outSockAddr has a valid initialized address, so that can be set to true.

From there, helper code is provided that allows the function to print out the recv'ed data for debugging. It then casts the buffer to a PDU and prints it out using the print_in_pdu() helper.

Finally the function returns the number of recv'ed bytes, if there were no errors.

<u>dpsend(dp_connp dp, void sbuff, int sbuff_sz)</u>
This function is simply just a helper for sending the dgrams. It checks to see if the send buffer size is larger than the max datagram size, and if it is, returns an error.

Otherwise, it calls the dpsenddgram() function, and return the number of bytes sent, minus the pdu size.

<u>dpsenddgram(dp_connp dp, void sbuff, int sbuff_sz)</u>
This function handles the actual sending of the PDUs and their payload.

It first checks both the outSockAddr, ensuring it is intialized, and then the send buff size once again.

If both checks pass, then it casts the dpBuffer to a buffer structure, sets the sendSz field to the sbuff_sz parameter, the proto_ver to Drexel Protocol, the message_type to `send`, the datagram size to the sendSize (which was set to the sbuff_sz earlier), and the sequence number to the sequence number stored in dp.

Next it copies the payload to the _dpBuffer, but uses to the size of the pdu structure to add the payload after the pdu.

It then calculates the total send size, by adding the payload and PDU struct sizes togther.

It then calls the dpsendraw() function, and the number of bytes sent is stored in a variable `bytesOut`.

The function checks to ensure all bytes were sent, and if not prints an error.

The sequence number is updated, if it sends an ack, then it is updated by 1, otherwise it increases to the size of the payload.

From there, the function looks for a recv'd acknowledgement, if it doesn't get the ack, it prints an error stating what type of mtype it recv'ed.

It then returns the payload size.

<u>dpsendraw(dp_connp dp, void sbuff, int sbuff_sz)</u>
This function is responsible for the actual sending of the PDU.

It first checks to ensure that the outSockAddr is initialied, and if not, returns an error.

It casts the send buffer to a pdu, and uses the sendto() function to send the buffer to the udp socket address.

It then prints out the pdu it sent, and returns the number of bytes sent.

dplisten(dp_connp dp)
This function acts as the listener in TCP.

It first checks to make sure that the connection address has been initialized, if not returning an error.

It then attempts to recv() a pdu, which will contain a connection message type. It this is true, it will prepare a pdu to send a connection acknowledgement message type.

It checks to ensure that the send was successful and that the size was correct.

Finally it prints that the connection was established and returns true.

dpconnect(dp_connp dp)

This function is responsible for setting up the dp connection.

It first checks to see if the server struct has been initialized and has an address. if not, returns an error.

Next, it attempts to send PDU with message type connect, and the datagram size of 0. It checks to ensure that the number of sent bytes is equal to the size of the PDU struct, if not returns an error.

After the sends, it recv's, checks the number of bytes recv'ed ensuring it is also the size of the PDU, and looks at the message type. Specifically, it looks for the message type CNTACK, meaning that the client and server can contact one another. If the mtype is not that, an error is returned meaning that the connection was unsucessful.

If it was successful, the sequence number is increased by 1, because we just recv'd an ACK, and the isConnected field is set to true.

The function then returns true, meaning that it is connected.

dpdisconnect(dp_connp dp)

This function is responsible for handling the proper disconnection between the client and server.

It first builds a PDU with the message type DP_MT_CLOSE, and has the datagram size set to 0.

It then attempts to use the dpsendraw() function to send the PDU, and checks the numbers of bytes sent to ensure it is equal to the size of the PDU struct. if it is not equal, it returns a general error.

It it was successful, the function then attempts to recv an ack. If the recv is not the size of the PDU struct, it returns a general error.

Otherwise, it looks at the message type to ensure it is of type CLOSEACK, otherwise it returns an error.

If it gets to the next line, it closes the connection and returns the DP_CONNECTION_CLOSED constant.

dp_prepare_send(dp_pdu pdu_ptr, void buff, int buff_sz)

The purpose of this function is to prepare a buffer to be used to send.

It checks to ensure that the buffer size is greater than the size of the pdu struct, if not returns NULL.

If it is, it clears the buffer (sets everything to 0) and copies the PDU into the buffer.

The function then returns a pointer to the location right after the PDU, so the payload can be added.

print_out_pdu(dp_pdu pdu)

This function is responsible for its name, printing the out PDU, but serves as a helper.

It checks to ensure that debug mode is enabled, if not it returns.

Otherwise, it calls the print_pdu_details() function, along with a header that states it is an outbound PDU.

print_in_pdu(dp_pdu pdu)
This function is basically the same as the previous function, except the header contains a statement that it is an inbound PDU.

print_pdu_details(dp_pdu pdu)
This function prints the version, message type, message size, and sequence number of a PDU struct. It is used in both print_out_pdu() and print_in_pdu().

pdu_msg_to_string(dp_pdu pdu)
This function is a helper functon that prints out a message type to easily readable english.