Ethan Weggel

erw74

PART ONE

/*

- dp_connp_dpinit() is a static function that exists only in the context of this file (du-proto.c).
- The goal of the function is to create a new instance of a dp_connection struct and initialize the values
- from random memory to useful starting values or zeroes. We start by declaring a variable 'dpsession' that is of
- type dp_connp which is typdef'd to represent a pointer to a dp_connection. This memory is pulled from the heap
- (however many bytes one dp_connection struct takes, denoted by sizseof(dp_connection)) and we save the pointer to this
- chunk of memory. We then initialize all bytes to the value of zero using bzero() and passing our pointer. Then for
- all fields we do the following:
- `dpsession->outSockAddr.isAddrInit = false [to say we have not intialized the address]`
- `dpsession->inSockAddr.isAddrInit = false [to say we have not intialized the address]`
- `dpsession->outSockAddr.len = sizeof(struct sockaddr_in) [to keep track of how big our 'outSock' address is]`
- `dpsession->inSockAddr.len = sizeof(struct sockaddr_in) [to keep track of how big our 'inSock' address is]`
- `dpsession->seqNum = 0 [to start our intial sequence number at zero when transmitting and receiving data]`
- `dpsession->dbgMode = true [to set our debug mode to true]`
-
- then we return this pointer so we can keep track of it and use it in other parts of our program with all of these fields
- ready to use in a neutral state.
  */

```
static dp_connp dpinit()
```

/*

- dp_close(dp_connp dpsession) simply takes an instance of dp_connp which is a pointer to a struct 'dp_connection'.
- This function then takes that pointer and frees the memory used to hold all fields and returns the memory to the heap
- so there are no memory leaks or resource problems in the program.
  */

```
void dpclose(dp_connp dpsession)
```

/*

- dpmaxdgram() simply returns a constant DP_MAX_BUFF_SZ. This is useful because DP_MAX_BUFF_SZ is used in other places
- to allocate memory to hold a datagram, but really, this also represents the maximum datagram we can store. It's a handy
- way to use one constant or 'magic number' and give it a second name within our program's context.
  */

```
int  dpmaxdgram()
```

/*

- dpServerInit(int port) at its core takes a port number and returns a server socket that is receptive to clients connecting
- (i.e.) it starts listening for connections. It starts by declaring variables for a sockaddr_in struct where we can store
- the address we are going to use for our server. Then we call our dpinit() function so that we get a new dp_connection struct
- that has all empty/neutral values. If there was an error getting this memory or setting any of the fields then we print an error
- and return from the function. Since servaddr and sock are both pointers, we can set the address both of these variables point to.
- Therefore, we point our sock pointer at our emtpy dp_connp field called dpc->udp_sock. This is also an int and serves as a replacement
- for an int that is external to our struct to hold our socket file descriptor. We do something similar with our inSockAddr.addr. We
- tell servaddr to now point to this place so that when we modify servaddr, we are really modifying the space that inSockAddr.arr
- occupies. Next, we create our actual socket using IPv4 over UDP rather than TCP; if there is an error in this process, then we error and

- return. Now that we have our socket, we need some information about how to bind the socket, so we set the servaddr (really the dp_connp)
- and say we are using IPv4, we set the port number and use INADDR_ANY. Then as a safety measure that is really helpful for debugging,
- we setsockopt so that the os will give us the port number back quickly versus letting it free the port all on its own. Then we bind the
- socket to this address and port using the bind() syscall and if there is an error we close the socket and error out. Finally we
- set the field in dp_connp to say the in-address is initialized and set the length of it equal to the size of 'struct sockaddr_in'.
- Finally we return this partially populated dp_connp.
  */

```
dp_connp dpServerInit(int port)
```

/*

- dp_connp dpClientInit(char *addr, int port) takes in a port and an address and returns a pointer to a dp_connection struct.
- In order to do this we declare variables for a 'struct sockaddr_in' which represents our server address, denoted 'servaddr';
- this is a pointer to this struct. We also declare a pointer to an integer that will represent our socket file descriptor.
- Once we declare these, the dpinit() function is called to make a blank dp_connection and return us a pointer to it; if there is
- an error, then we error out and return. Next we point sock at the dpc or dp_connp field udp_sock that is meant to hold an integer
- representing our socket filedescriptor. Then we point servaddr at 'dpc->outSockAddr.addr' so that when we modify servaddr, we are really
- modifying the corresponding field in our dp_connection. After this we create a socket by using the socket() syscall and we tell it to use
- IPv4 and UDP; if there is an error with our os syscall then we error out and return. Then after this we take our servaddr and set the address
- information to our server address (remember we changed where servaddr points). We set this up to be IPv4 using the port we pass in as
- a parameter and the address we pass in as a parameter. After this we set the length of the address in out dp_connection and say the out-address is
- initialized. We want the in-address to be the same as the out-address so we use memcpy() to copy the values of dcp->outSockAddr to dcp->inSockAddr
- for the length of the sockAddr. This also initializes the in-address. After this we return the pointer to the client dp_connection, denoted
- 'dp_connp'.
  */

```
dp_connp dpClientInit(char *addr, int port)
```

/*

- int dprecv(dp_connp dp, void *buff, int buff_sz) takes a pointer to a dp_connection, a pointer to a buffer
- and a size of that buffer. The function starts by declaring a new pointer to a dp_pdu and then serves as a
- wrapper for calling dprecvdgram(). We pass our pointer to our dp_connection, our global buffer for writing
- data '_dpBuffer' and the size of that buffer. This returns the number of bytes we received. If we received
- DP_CONNECTION_CLOSED as a result of dprecvdgram() then we return the same code. If this is not the case then
- we set the pointer to our dp_pdu to the beginning of our _dpBuffer which we wrote to. This points inPdu to the beginning
- of the received dp_pdu. If our receive size is larger than the size of our pdu, then we know that we have a payload on the
- other side of the pdu so we write that to our buffer that we pass in to our function. Finally, we return the
- full datagram size.
  */

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

/*

- static int dprecvdgram(dp_connp dp, void *buff, int buff_sz) takes a pointer to a dp_connection
- a pointer to a buffer and a buffer size. In essence, the goal of this function is to wrap our
- call to dprecvraw(). The function starts by declaring placeholders for how many bytes we've received
- and our error code. The function then checks to make sure the buffer size is not greater than the max size
- of our buffer (defined by a 'magic number' constant 'DP_BUFF_OVERSIZED'); if it is, we set the error code
- to inform the caller that the buffer we are writing to is oversized. Then we call the dprecvraw() function
- to receive the raw data and write it to the buffer we have, returning the number of bytes received; we error
- check and set the error code if applicable. Then we declare a new dp_pdu and copy the first part of the recv_buff
- (we only copy however many bytes are in a dp_pdu); we check for an error again and set the error code appropriately.

- Next we prepare the sequence number and our ACK. If we have an error, we simply increment the seq number by 1 and we are
- going to ACK our error. Otherwise, if the size we received was 0, this is a control message so we increment the sequence numer
- by one. If we don't error and its not a control message (just the pdu), we increment the sequence number by what is contained in
- inPdu.dgram_sz. After this, if we error'd on the previous step, we are going to send that error msg type and the ACK.
- If there is an error sending this, then we RETURN an error with the protocol. Then in the last section, if we have a send message
- type or a close message type, we simply send the appropriate messages and ACK's back to the sender rather than continue on. We
- then return the number of bytes we received in again.
  */

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

/*

- static int dprecvraw(dp_connp dp, void *buff, int buff_sz) takes in a pointer to a dp_connection,
- a pointer to a buffer and the size of that buffer. We first declare an integer to keep track of how
- many bytes we have received total. Then we see if our receive address or 'inSockAddr' is initialized,
- if not, we error out and return. After this, we are clear to start receiving bytes, so we make the call to
- recvfrom() where we use our binded socket address and our buffer and buffer size to receive data. We use
- recvfrom() and not recv() because this is a connectionless communication. We want to accept the message if and
- only if the address matches our outSockAddr (which we also specify). This returns to use the number of bytes
- we received and we update our integer 'bytes'. We also set outSockAddr.isAddrInit state to true because if it
- is null, then we fill that address space with that number of bytes (outSockAddr.len bytes) of the sender's address.
- We then have some debugging code which is hardcoded to be 'off' right now, but if we set our conditional to true
- then we will get a character pointer to our payload and then print the payload contents. Regardless of if this
- debugging code executes, we then print the incoming pdu contents. We finally return how many bytes we received.
  */

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

/*

- int dpsend(dp_connp dp, void *sbuff, int sbuff_sz) takes a pointer to a dp_connection, a pointer to a
- send buffer and the size of that buffer. The function starts by checking to see if our buffer size is bigger
- than the max datagram size; if this is the case, we return an appropriate error code. Otherwise we use this
- function as a wrapper to call dpsenddgram() and we return the number of bytes this subcall returns.
  */

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

/*

- static int dpsenddgram(dp_connp dp, void *sbuff, int sbuff_sz) take a pointer to a dp_connection, a pointer
- to our send buffer and the size of that buffer. The function starts by declaring an integer to store the
- number of bytes we send out. We then check to see if our outgoing address is initialized and if not we error
- and return an error code. If we do not have an error with the address, then we check to see if the semd buffer
- size is greater than the maximum buffer size we can send; if yes, we return an error code for a general error.
- If we get past our error checks, we start building the pdu and the buffer. We declare a new dp_pdu and point
- it to the start of the global buffer, _dpBuffer. We also set our send size equal to the buffer we passed in as an
- argument to the function. We then set the message type to a general send and then the dgram_sz equal to our send size.
- We also make sure to update this pdu's sequence to the most recently updated sequence number stored in our dp_connection.
- Then the function will copy the send buffer to our global _dpBuffer starting after the pdu and will copy the length of
- the send size (denoted 'sndSz'). To start an error check, we calculate the 'totalSendSz' by adding the datagram size with the
- size of the pdu. We then use this function as a wrapper to the dpsendraw() call; this will return how many bytes are sent. If

- the 'bytesOut' does not equal 'totalSendSz' then we have an error message, but we continue onward in our code. We then set the values
- of our pdu to zero so we can receive an ACK message. If we receive too few bytes to populate a pdu and our message type is not
- a message acknowledgement we write a new error message. Then we return how many bytes we sent out, minus how many bytes our pdu took.
  */

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

/*

- static int dpsendraw(dp_connp dp, void *sbuff, int sbuff_sz) takes a pointer to a dp_connection,
- a pointer to our send buffer and the size of that buffer. The function starts by declaring an integer
- to hold how many bytes we've sent. Then we check to see if the outgoing address, denoted by 'outSockAddr.isAddrInit',
- is initialized; if not, we error our and return an error code. If we are all good to go with the address, then we declare
- a new dp_pdu pointer and set the pointer equal to the beginning of our outgoing send buffer. We then pass our socket address
- to sendto() because we need the local address and the outgoing address since this is a connectionless communication protocol.
- We also pass the buffer with our pdu + payload in it and send it, storing the number of bytes sent in 'bytesOut'. We then
- print our outgoing pdu and return the number of bytes sent.
  */

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

/*

- int dplisten(dp_connp dp) takes a pointer to a dp_connection. We declare some values for our send size and our
- receive size. We also then check to see if our in-address is initialized; if not, we error and return a general
- error. Then we declare a new dp_pdu and then set all values equal to zero. We print a message indicating we are
- trying to connect and we call dprecvraw to see if any connection is trying to be made. If we are trying to set up a
- 'connection' then we should only receive the number of bytes needed for a dp_pdu. If the bytes received do not equal
- this then we error and return an error code. If we did receieve a connection pdu, then we denote this in our dp_connection

- field 'isConnected'. We then write a message saying we are connected and then we return 'true'.
  */

```
int dplisten(dp_connp dp)
```

/*

- int dpconnect(dp_connp dp) takes a pointer to a dp_connection. The function begins with declaring some integers
- to hold our send size and receive size. We then check to see if our outgoing address 'outSockAddr' is initialized.
- If we are not initialized we error and return an error code. If we make it past this check then we declare a new
- dp_pdu and set all the values to zero. We set the message type to connection and we set the current pdu sequence number
- equal to the most recent sequence number stored in our dp_connection. We then call dpsendraw() with our connection pdu
- and store how many bytes we sent in 'sndSz'. If our sent bytes don't equal the size of our dp_pdu, we know there was a problem
- so we error and return an error code. After this, we are expecting an ACK of sorts, so we call dprecvraw with our pdu to store
- the returning message. If we received anything other than the size of our pdu, we error and return an error code. Then we also
- check to see if the message type was a connection acknowledgment; if it is not, we error and return an error code. If we connected
- successfully then we increment our sequence number by one to denote a control transmission and then mark our dp_connection as
- connected. We then return 'true'.
  */

```
int dpconnect(dp_connp dp)
```

/*

- int dpdisconnect(dp_connp dp) takes a pointer to a dp_connection. Then we declare two integers to store our
- send size and our receive size. We then declare a new dp_pdu and intialize all of the values to zero. We
- set the message type to close and the pdu sequence number equal to the current sequence number stored in our
- dp_connection. We also say the dgram size is 0 since this is just a control transmission (pdu only, no payload).
- We then call dpsendraw() to send our pdu and store how many bytes were sent in 'sndSz'. If 'sndSz' does not match

- the size of out dp_pdu then we know that we did not send the full pdu and so we error and return an error code.
- Once we know we've sent the full pdu, we then are looking for an ACK, so we switch to receive dprecvraw() with our
- current dp_connection and our pdu. If the 'rcvSz' does not equal the size of our pdu then we know we did not receive a proper
- acknowledgement to our close request so we error and then return an error code (same deal if the message type is not a close
- connection acknowledgement). We then call dpclose() with our current dp_connection to free our memory. We then return an appropriate
- return code to signify that the connection is closed.
  */

```
int dpdisconnect(dp_connp dp)
```

/*

- void * dp_prepare_send(dp_pdu *pdu_ptr, void *buff, int buff_sz) takes a pointer to a populated dp_pdu struct,
- a pointer to our buffer and the size of the buffer. The function starts by checking to see if the buffer size if smaller
- than the size of a pdu and if it is, then we cannot populate the buffer so we error. After this we zero-out the buffer by
- using bzero() to turn the contents of buff for the length of our dp_pdu starting at the buff pointer. Then we copy over the
- contents contained in our pdu by passing the pointer of our destination (our buffer), the source (our pdu) and how much to copy
- over (the dp_pdu size). This then returns the pointer to the buffer, plus the length of the pdu. This now means when we write to the
- buffer next, we will have a populated pdu prepended to the data. This pointer also tells us where to start writing.
  */

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

/*

- void print_out_pdu(dp_pdu *pdu) uses a pointer to our populated pdu and checks to see if we have debug mode on. If we do not have it on, we return right away.
- Otherwise we print our header to signify we are SENDING a pdu and then call print_pdu_details to print the fields.
  */

```
void print_out_pdu(dp_pdu *pdu)
```

/*

- void print_in_pdu(dp_pdu *pdu) uses a pointer to our populated pdu and checks to see if we have debug mode on. If we do not have it on, we return right away.
- Otherwise we print our header to signify we are RECEIVING a pdu and then call print_pdu_details to print the fields.
  */

```
void print_in_pdu(dp_pdu *pdu)
```

/*

- static void print_pdu_details(dp_pdu *pdu) takes a pointer to a populated pdu and simply prints out the fields with proper headings or labels.
- This is so we can see what each pdu contains since the details remain the same whether the pdu is being received or sent out. print_pdu_details()
- uses pdu_msg_to_string to print the 'Msg Type' field.
  */

```
static void print_pdu_details(dp_pdu *pdu)
```

/*

- static char * pdu_msg_to_string(dp_pdu *pdu) simply takes in a pointer to our populated pdu and looks at the field
- called 'mtype' which is short for message type. These are growing powers of two so that each message type has exactly
- one bit on with the rest off in the byte that makes up mtype. We feed it into a switch statement using defined 'magic-
- numbers' in our du-proto.h file and then return string of what the message really is in human-readable english.
  */

```
static char * pdu_msg_to_string(dp_pdu *pdu)
```

/*

- This is a helper for testing if you want to inject random errors from
- time to time. It take a threshold number as a paramter and behaves as
- follows:
- `if threshold is < 1 it always returns FALSE or zero`
- `if threshold is > 99 it always returns TRUE or 1`
- `if (1 <= threshold <= 99) it generates a random number between`
- `1..100 and if the random number is less than the threshold`

- `it returns TRUE, else it returns false`
-
- Example: dprand(50) is a coin flip
- `dprand(25) will return true 25% of the time`
- `dprand(99) will return true 99% of the time`

*/

```
int dprand(int threshold)
```

PART TWO

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?

SEND LAYER(S) RESPONSIBILITY BREAKDOWN:

dpsendraw() - This function is responsible for using the sendto() function to actually send the data from our send buffer to the correct address

dpsenddgram() - This function is responsible for setting up the send buffer for dpsendraw() and handles acknowledgements

dpsend() - This function checks that we are able to send our whole datagram based on buffer size and then calls dpsenddgram()

RECEIVE LAYER(S) RESPONSIBILITY BREAKDOWN:

dprecvraw() - This function is responsible for using the recvfrom() function to actually receive the data into our receive buffer from the correct address

dprecvdgram() - This function is responsible for calling dprecvraw() and handling different message types as well as acknowledgements

dprecv() - This function is responsible for calling dprecvdgram() and then checks to make sure our connection
is or isn't closed, then copies any payload data to our receive buffer

I think that this is a good design, breaking it into three separate layers. The raw layer is responsible only for actually receiving the data and writing it to a buffer, the datagram layer is responsible for making sure we send or receive but handles some higher level transformations of message types and

acknowledgements and some error handling. The highest level is responsible for providing some high level error checks and is a good clean interface for handling the middle layer calls. I wouldn't make any changes primarily because the function calls made are abstracting away complications and the relevant calls are all similar to that layer's level of abstraction (i.e. the lowest level 'raw' layer makes syscalls and the highest level calls our middleware). The error messages are also relevant to the functions being called at that level (i.e. it wouldn't make sense for the highest or middle layer to report a syscall error, etc).

PART THREE

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)?

Sequence numbers are used in du-proto, at a high level, function as one would expect sequence numbers to behave. They signify that the sender or receiver have received all bytes up to the given ACK. However, the way we pass sequence numbers around keeps things modular. Here we keep track of the current sequence number on either side of the UDP 'connection' in the dp_connection struct. Then whenever we send a message, we take that current number and place a copy of it in the dp_pdu that we intend to send out to the opposite side.

We update the sequence number for things that must be acknowledged like ACK responses because in the du-proto we are adding some bits of reliability on top of UDP which is inherently unreliable. We are able to match the ACK to the exact pdu that it is acknowledging and we are tracking exactly how much the receiver has consumed. Because we know what to expect, we can rule out duplicates or resends that we do not need or out of order datagrams which UDP can do sometimes.

PART FOUR

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?

One example limitation of this UDP du-proto approach of waiting for an ACK before the next send is allowed is that is dramatically reduces throughput compared to TCP. With TCP, the sliding window allows for many sends and a single ACK can account for those many sends. With this approach we need to wait for the ACK for a single send which can fluctuate on network conditions, congestion, server conditions and connection strength.

This *does* make some dramatic simplifications to this implementation, however. Considering we are waiting for an ACK before we send our next PDU + datagram, we no longer need a sliding window, our

sequence numbers simply track how many bytes have been acknowledged, we don't need reassebly algorithms at play and error handling is trivial (i.e. if and ACK doesn't arrive we just retransmit a single packet and don't need to maintain multiple timers.)

PART FIVE

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

Setting up TCP connections versus a UDP one have some pretty discernable differences. With TCP, since we need a three-way handshake to set up a connection, we need to call connect(), listen() and accept() on our server to form that connection. With UDP we need no such handshake and the server simply calls bind() and we use sendto() and recvfrom(). This is far more trivial when manifest in code.

It is worth making the further distinction that since UDP does *not* form a connection we use sendto() and recvfrom() rather than send() and recv(). This ensures that we know the sender and receiver with every operation.

The other thing to mention is that TCP uses streams and UDP sends and receives exactly one datagram at a time. With UDP we have clear message boundaries and the way we receive and send things works a little different. This means that TCP's recv() might return fewer bytes than sent, more bytes combined from multiple sends and it might require more in-depth parsing to get the same level of coherence between distinct messages.
