We started our project by doing steps 1 and 4 first to test TCP on our own machine as well as to the server, and this ended up taking quite a bit of time due to differences in build environment as well as issues with the server, which we discussed with both Dr. Christensen and our TA Steven. Most of this was resolved through proper port selection and using the IP of the server rather than the standard URL, and then we could start the coding of UDP to act like TCP. On a single machine from console to console, the speed was around 8s for the 100MB and to the server it was around 30s or so.

For our protocol design, we checked on the internet through a few sources, the textbook, peers, as well as asked the Dr. Christensen for advice, and we ended up using Stop-and-Wait (SAW). We understood due to the nature of the protocol, it would be slower than Go-Back-N (GBN) and Selective Repeat (SR), which were the other two automatic repeat request (ARQ) protocols that use sliding window. All three protocols were discussed in the lecture, but we also went online to search for more information about them. From discussion with some other groups that started the project early, we learned that SR was far too difficult to implement properly and some of the groups had to start over with a different protocol due to that, which ended up costing time. For the group that did get some sort of SR working it also seemed that their speed was extremely slow, so we might as well just go with SAW to save time, then optimize the code to try to get it close to TCP speeds after the code was working Our professor also advised us to use SAW due to the simplicity of it, even after telling us to look at UDT and NETBLT first (these were even more complicated than SR).

Additionally, to make UDP reliable like TCP, there were a few things to consider and various ways to implement the protocol. Packet size could not exceed network maximum transmission unit (MTU), we should probably consider latency for speed, we could add sequence number to each packet, could use gap detection and gap fill to fix errors, and we can also use ACKs like TCP to make sure all frames are received. We were also not supposed to use any WSA socket function (Windows only) since the server would be running on a Linux distribution.

Some things to note is that SAW is better on a low link speed than high speed link [3], so it isn’t too appropriate for this project since we are operating at over 50Mbps and transferring a 100MB file. Source three had some calculation but basically on a 10km link and low speed of 50kbps, the channel utilization would be 99.94%, while on a high speed 300Mbps link, channel utilization would only be 21.07% [3]. However, our priority in this project is first working code then to worry about speed. We increased the buffer for UDP to 60000 and now the speed is like TCP. This is near the max UDP size which is 65,607

Here is a basically how the SAW protocol would work in our UDP transfer.

SAW Sender:

1. Open + bind socket
2. Open file for reading/sending
3. Send one frame
4. Wait for acknowledgement
   1. Start timeout clock
   2. If timeout clock expires, send frame again
5. Go back to 3 until file finishes sending

SAW Receiver:

1. Open + bind socket
2. Open file for writing
3. Receive frame
   1. Send ACK if frame is valid
4. Write data to file

gcc udp\_recv\_file.c sw.c -lws2\_32 -o recv  
gcc udp\_send\_file.c sw.c -lws2\_32 -o send

sw.c and sw.h : implementation of the stop n wait protocol

sw.c has static and non-static functions. The static functions are used only inside the sw file. Non-static functions have prototypes in the header and can be called from other files (encapsulation).

Receiver and Sender has all networking stuff:  
Creating socket  
Sending data  
Receiving data

The SW implements the Stop and Wait protocol.

In sender.c we create the socket and implement the functions that are used for:  
- reading data from file  
- sending data to receiver  
- receiving data from the receiver (for receiving ACK packets)  
- timer function, which returns 1 if there is incoming data from the receiver(wait for ACK packet),  
and returns 0 - if the timer expires and there is no message from the receiver  
  
After preparing all the sockets (same as given udp)

1. allocate memory for the sw\_ctx\_t - the structure that holds all the data of the SW protocol. This is done by calling function sw\_init.  
2. set callbacks. That means “if you need to send data, call this function"  
 data needed is sent

For example send function

static int send\_data(void \*data, int data\_len, void \*ctx)  
{  
int ret\_code;  
transport\_info\_t \*transport = (transport\_info\_t \*)ctx;  
  
ret\_code = sendto(transport->socket, data, data\_len, 0, transport->addr,  
transport->addr\_len);  
  
return ret\_code;  
}  
It uses sendto() to send the data. And sendto() require the address of the receive, the length of this address and the socket. All this data we put to the structure below:

typedef struct  
{  
int socket; //socket which we use to send data  
struct sockaddr \*addr; //receiver address  
socklen\_t addr\_len; //address length  
} transport\_info\_t;

We need to put all this data to one structure because have only one argument to be passed to the callback set - void pointer.

sw\_set\_cb(sw\_ctx, SW\_SEND\_CB, send\_data, &transport);  
SW\_READ\_DATA, //read data from file, set state to send packet  
SW\_SEND\_PACKET,//send packet, then go to waiting state for an acknowledgment  
SW\_RECV\_ACK, //wait For an acknowledgment, if ACK received - read next chunk of data  
if no ACK received, resend the previous chunk of data.  
SW\_COMPLETE, //the last chunk of data was read and sent  
and ACK received from the receiver   
SW\_ERROR, // there is an error  
  
There are two types of packets there:  
sw\_packet\_t  
typed struct  
{  
int seq; //the number order of data chunk we are sending  
int buff\_size; //the length of the chunk  
char buffer[BUFF\_SIZE]; //data  
} sw\_packet\_t;

The sequence (seq) is used to make sure that the data will be received in the same order that it was sent.

At start, receiver waits for packet 0 and sender sends packet 0  
When the sender receives ACK for 0 packet, it increments the sequence. And start sending the second packet. Process continues

Error handling  
There can be two bad scenarios  
S - sender, R - Receiver  
1. S sends packet 0, R - waits for packet 0  
problems during the transmission, the packet was lost.  
S waits 10 seconds,does not receive ACK from R, and resend the packets.  
  
2. S sends packet 0, receiver waits for packet 0  
R received the packet and sent back the ACk  
ACK was lost, and S didn't receive it  
The Sender thinks that receiver did not receive the data and starts the sending progress again. Receiver receives package sequence 0 now which it has already received and it is waiting for packet with sequence 1. Receiver discards the data and send ACK for 0 packet to the sender  
again.  
The sender receives the ACK and starting to send chunk #1.  
  
For reading data from the file, the application uses POSIX C API  
For transfer data through the network is used Berkley Sockets API, via UDP protocol.  
To pass all the data that is required by sendto() and recvfrom() function, the data is encapsulated to structure transport\_info\_t that contains socket, network address and the length of this address.

sender: read>send>wait for ACK>ACK received > read next > send and so on  
receiver: recv>write>send ACK > recv > write and so on

Citations

1. <https://en.wikipedia.org/wiki/Automatic_repeat_request>
2. <https://en.wikipedia.org/wiki/Stop-and-wait_ARQ>
3. <https://en.wikipedia.org/wiki/User_Datagram_Protocol>
4. <http://www.mathcs.emory.edu/~cheung/Courses/455/Syllabus/3-datalink/stop-and-wait-anal.html>
5. <http://www.pcvr.nl/tcpip/udp_user.htm>