Exercise1 documentation

Name: Wei Huo JHED: whuo1

1. Packet Structure
   In my implementation, I designed a packet header with the following format:

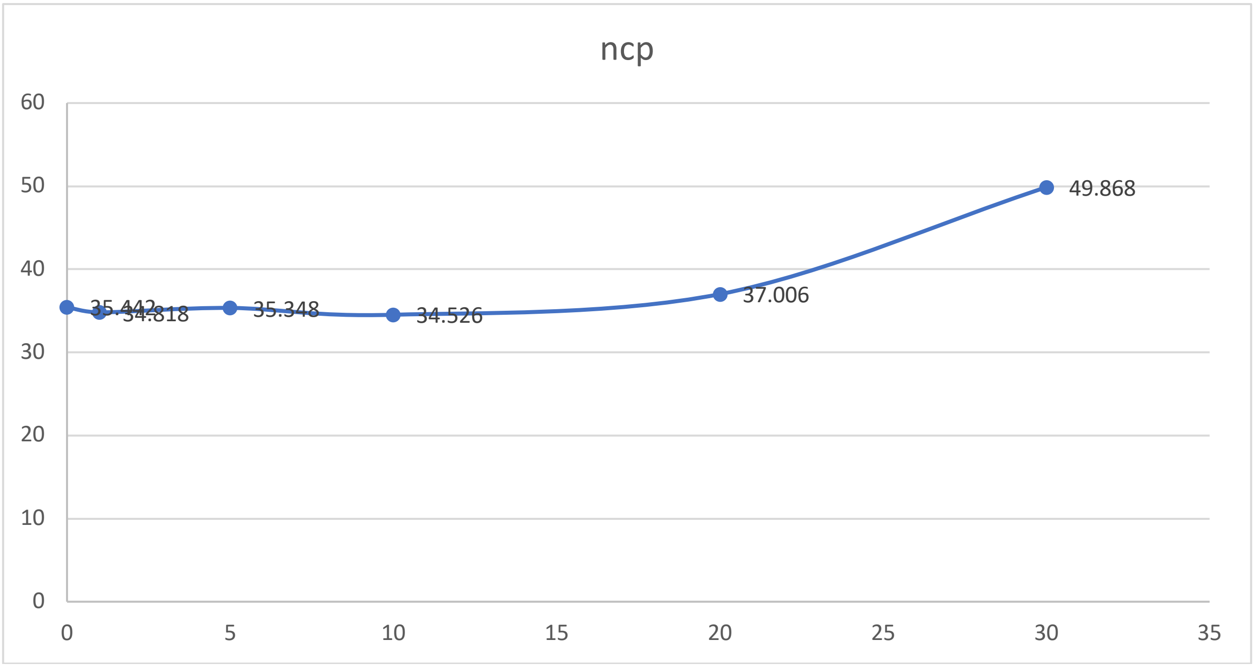
```
struct Packet{
    unsigned long numPacket;
    unsigned short dataSize;
    enum Status status;
    unsigned long seqNum;
    char data[SLOT_SIZE];
    char filename[20];
}
numPacket: the total number of packets that the receiver is supposed to
receive.
dataSize: the size of the data wrapped in the packet.
status: a few options including SYN, SYNACK, SEND, ACK, NAK, WAIT.
seqNum: the sequence number of the packet.
data: contains the splitted data of size 1000bytes.
filename: the name of the file that the receiver is supposed to write to.


SYN: the client send to the server in order to detect if the server is busy.
SYNACK: if the server is not busy, it will send SYNACK to the client and the
client can start sending data.
SEND: a packet sent from the sender.
ACK: a packet sent from the receiver informing the sender that all the packets
up until the seqNum are received.
NAK: the packet with seqNum is not received
WAIT: the server is busy, the sender has to wait.
```

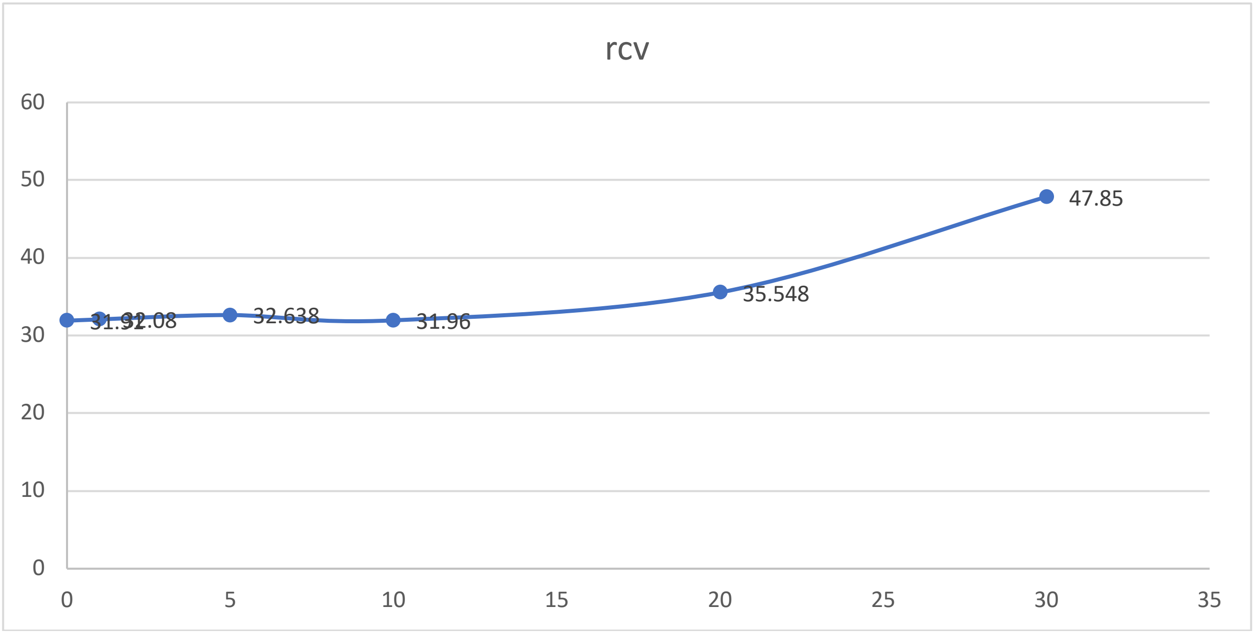
2. Protocol.
   The protocol runs in the following manner:

a. The file was splitted into chunks of size 1000 bytes and stored in an array, the sender will send each chunk in order.
b. The sender sends a SYN packet to check if it can send data, if SYNACK is received, it will start sending data, otherwise, if WAIT is received, it has to wait for 2 seconds. If the server is still busy after 2 seconds, the sender will double the time for waiting, which is 4 seconds in this case, and so on. Once the sender starts sending data, the timeout will be reset to 2 seconds. This is somewhat like the exponential backoff.
c. The way we detect if the server is busy is by comparing the hostname of the senders. If a sender with hostname x is currently communicating with the server, then the server will keep its hostname in the loop. Once another sender with hostname y is trying to send, the server will see that x is not equal to y such that it will send a WAIT packet to the second sender to make him wait.
d. For every packet with status SEND sent from the sender, the server will respond with an ACK. We use cumulative acknowledgement. We have a window of size 100 for both the sender and the receiver. The window size is chosen after careful experiments. If the window size is too large, a severe packet loss will be observed. If too small, the transmission will be slow. Under any circumstances, there will only be 100 packets in transmission. Once all the packets are received by the receiver, it will ack with the next expected sequence number, the sender will start sending the next 100 packets. If the receiver found any missing packet with the sequence number below the most recent packet it received, it will send a NAK with the corresponding sequence number to the sender, and the sender will resend immediately. The window is implemented with an array of size 100, using index from 0 to 99 to denote the corresponding packet. Let's say if we have a base sequence number x, and we have received another packet with sequence number y. If y is less than x, we simply ignore it since it's an old packet which we have already dealt with properly. If y >= x, we will get the index y-x and set window[y-x]=1 to indicate that the corresponding data has been properly received. We are confident that y-x would not exceed 100 since we will not send packet with sequence number larger than x+99. Once all the window slots are labeled with 1, we will update the base sequence number with x+100, clear the window labels, and continue until we have received or sent all the packets.
e. If the sender or the receiver didn't get a packet after a 2 seconds timeout, they will resend the corresponding packets. The timeout is as important as the window size. Experiments told me that smaller timeout will incur more resends, while larger timeout will make both parties wait for a long time.
f. Once all the packets are received, the sender will do a cleanup and shut down. The receiver will write all the data received into a file with a specified file name, do the cleanup and be ready for the next client.

3. Results and discussion.
   a. The loss rate versus transmission time graph for ncp and rcv when sending 1GB binary file.

As we can see, the transmission times are relatively consistent for loss rate 0%, 1%, 5%, 10%. This is because the loss rates we configured are just effective for the sender and the receiver, but not for the channel. The UDP/IP channel has an even larger loss rate that somewhat hides the loss rates we configured. However, when the loss rate is configured much larger, say 20%, 30%, the transmission will be dominated by the loss rate we configured, thus showing the expected curve as drawn below.





b. The 5 duplicate experiments for different loss rate and the average outcome when sending 1GB binary file.

NCP

loss rate(%)	file trans rate(Mbits/s)	time cost(seconds)
0	216.78	36.9
	222.96	35.88
	234.84	34.07
	221.14	36.18
	234.08	34.18
avg	225.96	35.442
1	230.93	34.64
	218.78	36.57
	250.97	31.88
	222.61	35.94
	228.19	35.06
avg	230.296	34.818
5	223.35	35.82
	222.5	35.96
	226.38	35.34
	242.57	32.98
	218.33	36.64
avg	226.626	35.348
10	226.68	35.29
	229	34.93
	237.09	33.74
	234.21	34.16
	231.79	34.51
avg	231.754	34.526
20	228.07	35.08
	219.13	36.51
	219.82	36.39

	225.42	35.49
	192.47	41.56
avg	216.982	37.006
30	159.81	50.06
	161.54	49.52
	157.2	50.89
	161.28	49.6
	162.36	49.27
avg	160.438	49.868

RCV

loss rate(%)	**file trans rate(Mbits/s)**	**time cost(seconds)**
0	244.24	32.75
	243.89	32.8
	251.15	31.85
	251.12	31.86
	264.15	30.29
avg	250.91	31.91
1	238.22	33.58
	244.25	32.75
	274.03	29.19
	244.65	32.7
	248.57	32.18
avg	249.944	32.08
5	240.55	33.26
	257.01	31.13
	243.5	32.85
	243.27	32.89
	242	33.06
avg	245.266	32.638

10	258.71	30.92
	244.84	32.67
	245.45	32.59
	247.87	32.28
	255.29	31.34
avg	250.432	31.96
20	224.45	35.64
	225.58	35.46
	231.46	34.56
	223.42	35.81
	220.59	36.27
avg	225.1	35.548
30	168.88	47.37
	165.27	48.41
	169.45	47.21
	171.65	46.61
	161.14	49.65
avg	167.278	47.85

c. The transmission rate and time for t_ncp and t_rcv when sending 1GB binary file. The parallel running of t_rcv/t_ncp and rcv/ncp is also tested. But the transmission efficiency gap is so dramatic such that we cannot expect a fair game.

	File trans rate(Mbits/s)	**time cost(seconds)**
t_ncp	4569.84	1.75
t_rcv	4470.66	1.79
t_rcv (with rcv)	4319.27	1.85
Rcv (with t_rcv)	211.95	37.75
t_ncp (with ncp)	5501.87	1.45
Ncp (with t_ncp)	197.04	40.6

4. Conclusion.

To conclude, the protocol I designed is way less efficient than the TCP/IP protocol. There are a few strategies that could greatly improve the transmission efficiency but I don't have time to implement.

a. Include more NAK sequence numbers in the packet such that there will be fewer packets looping around.
b. Take full advantage of 1400 bytes packet size which is guaranteed not to be splitted up during the transmission.
c. Try to lower the timeout. This value is quite contingent upon the entire protocol design. In my current protocol, I have to make it relatively large because I have already got a lot of packets looping around, I cannot afford more timeout packets, which will cause more severe packet loss. If we could include more NAK sequence number in a packet, the number of the packets looping will be dramatically decreased, such that a lower timeout will be more useful.