# Distributed Systems Exercise 1 Design Document

Minqi Ma (JHED: mma17), Jerry Chen (JHED: gchen41)

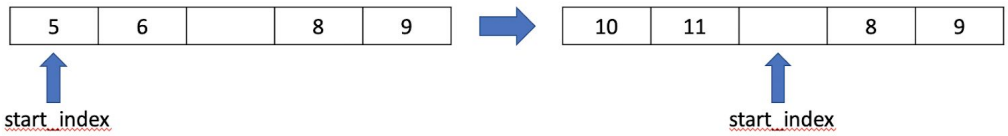
## Main Idea

- Use Selective Repeat ARQ
- Receiver sends back cumulative acknowledgement (ack) and negative acknowledgement (nack), which specifies indices of missing packets
- Sender continuously slides window and send new packets as ack grows, and conduct retransmission at nack. It also sends the last packet at timeout.
- Packets have special tags in the beginning to specify start/middle/end of one transfer.

## Internal Data Structures

The sender process (ncp):

- An array to represent window, which store WINDOW_SIZE of packets before sending
    - Note: when sliding the window, we wrap around the array and store the new "start" index of the array, instead of shifting all the entries in O(n).
      For example: shift window for 2 cells; start_index becomes 2; new packets will be filled in at index 0, 1




    - We are using this technique for *every* array structure below.

The receiver process (rcv):

- An array to represent window, which stores WINDOW_SIZE of packets.
- An array of boolean to indicate if each entry in the window is occupied or not
- An array of struct timeval to represent timestamp of each sent nack in the window

## Type and Structure of the messages

struct packet (packet sent from ncp to rcv)

- unsigned int tag; // identify if packet contains filename, file data or is the last packet
- unsigned int sequence; // sequence number of the packet
- unsigned int bytes; // number of bytes in the file data
- char file[BUF_SIZE]; // file data

struct packet_mess (packet sent from rcv to ncp)

- unsigned int tag; // identify if packet signals start, busy, or end transfer
- unsigned int ack; // cumulative ack
- unsigned int nums_nack; // number of nacks
- unsigned int nack[NACK_SIZE]; // sequences of packets that rcv misses
-

tag

- tags sent by ncp
    - NCP_FILENAME 0 // packet includes filename
    - NCP_FILE 1 // packet includes file data
    - NCP_LAST 2 // sent the last packet with file data
- tags sent by rcv

- RCV_START 0 // rcv is not busy and can start transfer
- RCV_ACK 1 // packet with ack and nack value
- RCV_BUSY 2 // rcv is busy and cannot connect
- RCV_END 3 // rcv has received the last packet and end transfer

**Protocol description**

ncp:

1. Send first packet with filename to issue connect request
    a. If rcv responds with RCV_BUSY, wait for 10 seconds and send the same request again
    b. If rcv responds with RCV_START, continue
2. Read WINDOW_SIZE of packets to the window, and send them in sequence.
3. When rcv responds with RCV_ACK
    a. Read ack value and slide the window accordingly.
    b. If there is any nack, retransmit nack packets.
4. If does not hear any response after a certain amount of time, send the *last* packet in the current window.
5. When read the last packet into the window, give it NCP_LAST tag.
6. When rcv responds with RCV_END, exit the program.

rcv:

1. Receive a packet with NCP_FILENAME
    a. If currently free, send back RCV_START
    b. If currently busy, send back RCV_END (it might be previous client who misses RCV_END packet)
2. Receive a packet with NCP_FILE
    a. Put the packet in the corresponding spot in the window
    b. Slide the window up to the first gap; it will be the cumulative ack value sent back
    c. If there is any missing packets in between start of window and last received packet
        i. If there is no timestamp for nack in this cell, include nack in the return packet
        ii. If there is timestamp, check the time difference; if the time difference is large, include nack in the return packet, else not.
    d. Send return packet with tag RCV_ACK
3. Receive a packet with NCP_LAST, record the last sequence. Perform as step 2.
4. If cumulative ack value == last sequence, send RCV_END, and mark itself as free.

Constant:

1. The max number of nack is defined to be 50 to reduce the pressure on the sender side at a high loss rate.
2. Packet size is set to be 1400 bytes.
3. The length of the destination file name needs to be <= BUF_SIZE, which is the size of file data segment in the packet. Our BUF_SIZE = 1400 - 3 * sizeof(int) = 1388 bytes.
4. ncp detects the loss rate dynamically, and adjust timeout interval accordingly. We round up the loss rate to the closest level and change the interval as shown in the table below:

| loss_rate_percent (%) | timeout_interval_usec (ms) |
|---|---|
| 0-5 | 6000 |

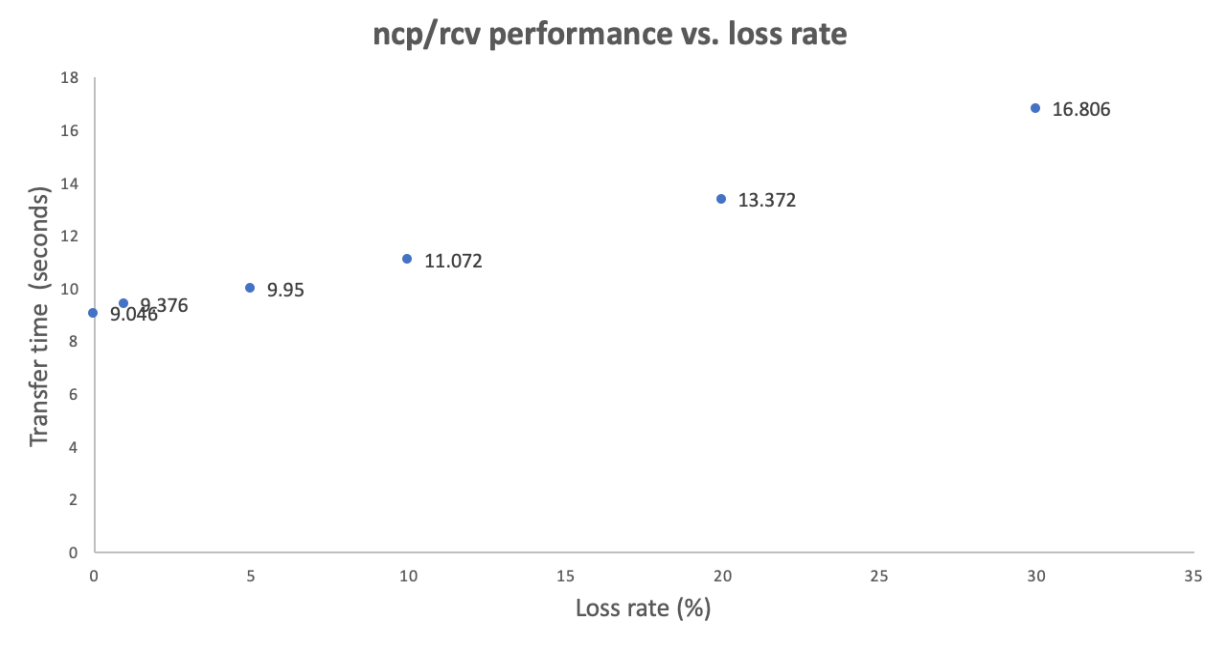
| | |
|---|---|
| 5-10 | 3000 |
| 10-20 | 2000 |
| >20 | 1000 |

**Performance**

Result of test trials

| | 1 | 2 | 3 | 4 | 5 | AVG |
|---|---|---|---|---|---|---|
| t_ncp/t_rcv | 8.98 | 8.98 | 8.98 | 8.98 | 9.01 | 8.986 |
| ncp/rcv with zero loss rate | 9.02 | 9.06 | 9.1 | 9.06 | 8.99 | 9.046 |
| ncp/rcv with %1 loss rate | 9.39 | 9.35 | 9.39 | 9.34 | 9.41 | 9.376 |
| ncp/rcv with %5 loss rate | 9.96 | 9.95 | 9.93 | 10.01 | 9.9 | 9.95 |
| ncp/rcv with %10 loss rate | 10.98 | 10.99 | 10.85 | 10.84 | 11.7 | 11.072 |
| ncp/rcv with %20 loss rate | 13.23 | 13.31 | 13.72 | 13.27 | 13.33 | 13.372 |
| ncp/rcv with %30 loss rate | 16.82 | 16.45 | 17.21 | 16.93 | 16.62 | 16.806 |
| ncp/rcv with zero loss rate, run AT THE SAME TIME with t_ncp/t_rcv | | | | | | |
| ncp/rcv | 18.4 | 18.24 | 17.69 | 17.57 | 18.06 | 17.992 |
| t_ncp/t_rcv | 14.37 | 14.42 | 14.63 | 13.86 | 15.95 | 14.646 |

Graph of ncp/rcv performance vs. loss rate



**Discussion**

- As the graph shows, the UDP performance decreases as the loss rate increases. We tried to reduce the effect of increased loss rate by dynamically detecting the loss rate and setting corresponding timeout intervals on the ncp side. The results are reasonable to us, since the time ranges from 10 to 17 seconds, and the difference is tolerable.
- TCP overall is the fastest compared to UDP. It means that TCP protocol using data stream is significantly better than UDP using data segment. It saves the time of sending/receiving packets back and forth, dealing with missing packets, etc.
- When TCP and UDP are running at the same time, TCP runs faster than UDP and finishes earlier. It shows that TCP gets to use more resources than UDP. For they are competing at the same time, their performance is relatively worse than their normal performance.