Exercise 2 Final Design
Distributed Systems 600.437

Huabin Liu hliu76@jhu.edu
Qianyu Luo qluo3@jhu.edu

We implemented a multicast tool using Single Ring protocol achieved agreed order.

Structure:

start_mcast:

This program only multicasts a special char array to let every machine know that everybody is joined in.

mcast:

We created three structs as following:

packet:
type: PACKET_TYPE
machine_index
packet_index
random_num: a random integer from 1 - 1000000.
payload[PAYLOAD_LEN]: 1300 bytes payload.

token:
type: TOKEN_TYPE or TERM_TYPE (only for last round)
seq: the largest packet number sent out
aru: the largest packet number received consecutively.
random: used to distinguish duplicate tokens when it's impossible to distinguish only according to seq
max_remain_pack: the max remaining packets of a single machine
rtr[WINDOW_LEN]

init:
type: INIT_TYPE
sender_index: the machine index of this init message
sender_host[NAME_LEN]: sender's hostname
receivedNext: if sender received its next machine's hostname.
receivedByLast: if sender's hostname received by its last machine.

Protocol:

1. After some fundamental settings, call recv_dbg_init() function to initialize recv_dbg. Then call start(), waiting for the starting message to get permission going to next step.

2. Call initRing () function to build up the ring.

While in the function initRing (), firstly each machine builds its initPacket. And then begin to send initPacket by multicast. During the period of for loop, we use two integers to track if the received initPackets came from the next machine or the last machine. If from the last machine, check if last machine has received my initPacket. And if from the next machine, it should store its hostname and set receivedNext = 1.

After that, if a machine's receivedNext = 1 and receivedByLast =1, it means this machine could break. Otherwise, this machine would keep multicasting its initPacket. To ensure that machine 1 is the last one getting out from initRing function, machine 1 can only break when timeout.

When the initRing() finishes, each machines get the IP from its next neighbor.

3. After returned from initRing (), get into the main "for" loop. In the whole process, the number of packets sending out by a machine per round cannot exceeds constant MESS_PER_ROUND, meanwhile, a single machine can send no more than its total packets. We ask machine #1 to start sending packets and initialize a token. Then the token is transferred one by one.

AS A SENDER:

Machine who gets the token should first renew token aru according to the following rules:

1) If token aru is larger, it lowers token aru to local aru and and set last_change_machine to its own machine index.
2) If token aru smaller than local aru and last_change_machine is itself, change the aru to its local aru.

After updating token aru, the program check token rtr. It sends out all the rtr if it has that packet in buffer, deleting the corresponding rtr as long as sending out. Then check its own buffer and add its own rtr to token rtr.

At the beginning we thought something like linked list should be good idea to store rtr and we implemented linkedlist.h by ourselves, however, we find it doesn't work since transmission are among different machines. So we need to use array with constant length. Considering the number of rtr should normally be much less than window length, we would like to find a way to scan rtr array efficiently. The array is initialized to be -2. We insert rtr from the head of array, putting the rtr's packet number into array. When deleting a rtr, set that slot to be -1. Then when adding new rtr, start checking from the head of the array, when there is an empty slot (-1 or -2), insert the element there. By this way we try to maximize the utilization of the beginning part of the array and try to limit all the rtr within a comparatively short range.

Next, the program check whether it's allowed to send new packets according to the following conditions:

1) Whether the window buffer is full or not. If full, it cannot send.
2) Whether it has more packets to send or not. If no more, it cannot send.
3) Whether it has reached the max number of packets allowed to be sent in that round or not. If reached, it cannot send.

If token aru = token seq = local aru, increase aru together with seq when sending new messages.

Compare last token aru with current token aru (before updating), packets with index up to the smaller number in them can be delivered. Increase deliver_seq correspondingly.

Change to unicast mode, generate a new random number replacing the old one in token buffer, and send token to next neighbor in the ring.

If timeout, each machine should send out its latest token to next neighbor. The duplicate token can be judged by receiver.

AS A RECEIVER:

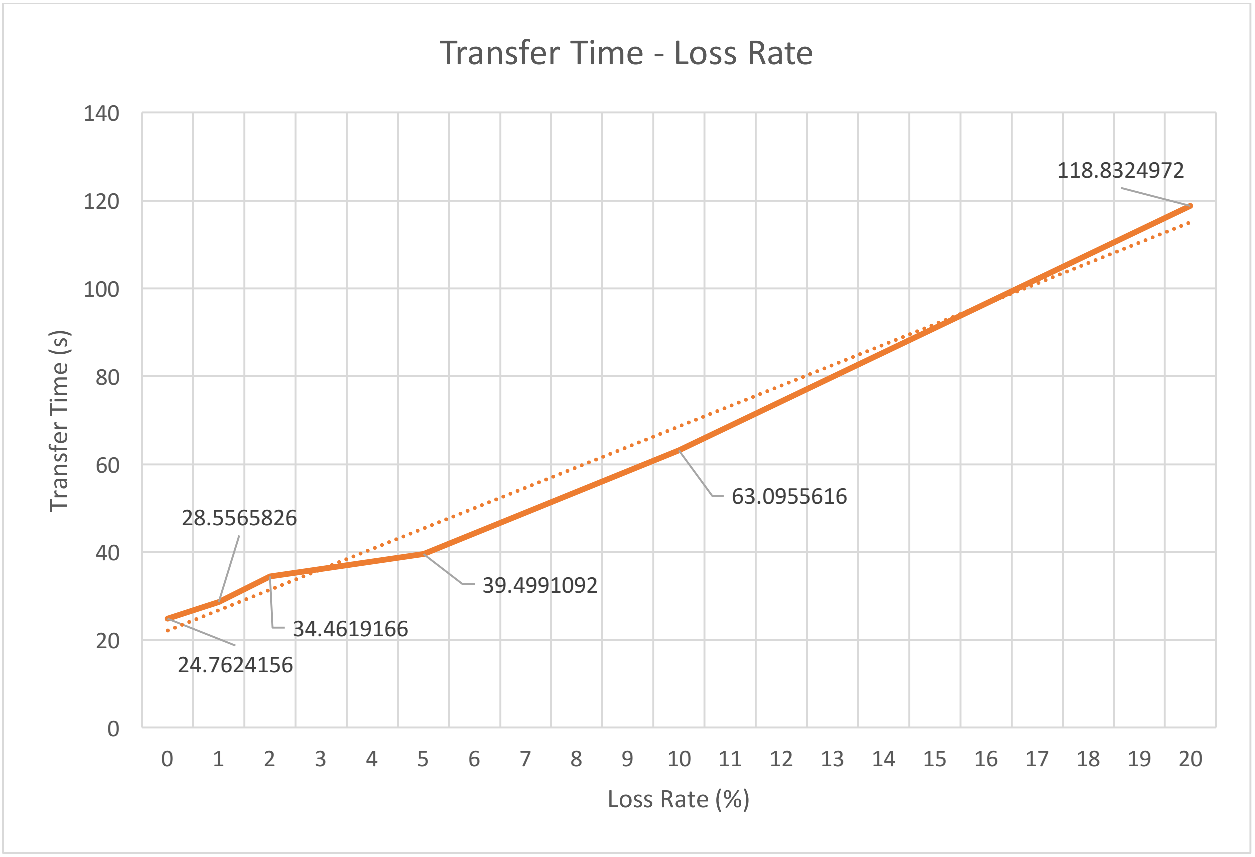
First check whether the packet is in the range from deliver_seq (first packet pending deliver) to local sequence. If so, it should be a rtr of certain member, put it to window buffer directly. If not, check whether the packet index fit in window buffer or not, if fit, put the packet into buffer and increase local sequence correspondingly. Otherwise drop it.

When receive a token, first check the token sequence. If it's smaller than current token seq, it must be a duplicate one, simply drop it. If it's larger, it must be a legal one. Then the program should put the token into its token buffer and start acting as a sender.

It's confusing only if token seq equals to current token seq. Then, we need to compare token random number and token aru, if both of them equals to old one, that should be regarded as a duplicate one. If not, we should consider possibility of termination. Check the following variables: local aru, last_token_aru, this token aru, this token seq. If they are all the same and if max_remain_pack = 0, means no one has packets to send, no one has rtr to request, nothing happened in the last round, everybody is able to terminate. So this machine should alter type of its token to TERM_TYPE, and multicast it to others indicating that it's time to stop. Any machine receives this termination token should also multicast it and deliver all the packets remain in buffer, close the file, then break out.

Discussion:

Our test results:



Transfer Time - Loss Rate Test

Loss Rate	1	2	3	4	5	Avg (s)
0	27.802674	22.879411	23.431006	27.761204	21.937783	24.7624156
1	27.23587	28.72648	28.81227	28.512164	29.496129	28.5565826
2	31.120928	36.364631	35.510255	39.035623	30.278146	34.4619166
5	40.177013	40.413532	38.330213	38.713851	39.860937	39.4991092
10	63.002725	67.093204	63.511804	60.964662	60.905413	63.0955616
20	121.027803	116.493841	116.418177	117.062136	123.160529	118.8324972

According to the graph above, we find that the average transfer time linearly grows with loss rate. However, the real transfer time is significant influenced by network condition. During our tests, when running the program under the same loss rate, we find that sometimes when the network seems to be not really stable or ugrad machines are too crowded, the transfer time difference can get up to 40% - 50%.

Parameters:

Time out: 1800us
Message per round per machine: 20
Window length: 344

After a lot of tests and comparisons, we find that above parameters make our protocol run fastest.