Documentation for exercise 2

Name: Wei Huo

The protocol I used in the final implementation is token-based one. I implemented the first edition based on my initial design but found intolerable inefficiency. I will talk a bit about my original design but will experiment with the token-based protocol.
In the token-based protocol. I designed a token struct and a packet struct.

```
struct Packet{
    int machineIndex;
    int packetIndex;
    int randomNumber;
    int valid;
    char payload[MAX_MESS_LEN];
};

machineIndex: index of the machine
packetIndex: index of the packet
randomNumber: a randomly generated number to differentiate packets
valid: 1 if the packet contains useful info, otherwise 0
payload: not used, but to satisfy the requirement on the size of the packet


struct Token{
    int next;
    int aru;
    int rtr[11][WINDOW_SIZE];
    int seq;
    int round;
    int tokenRound;
};

next: the machineIndex to receive the token
aru: the first unacknowledged packtet index in order
rtc: a 2D matrix with machineIndex as row and window indices as column, to denote
which packet needs to be retransmitted.
seq: the largest packetIndex within the current window size
round: a helper field to differentiate token passing stages
tokenRound: a helper field to prevent machine from receiving the token twice in a
single round when the token is missing and retransmitted
```

The token based protocol runs in the following way:

1. Start_mcast sent the token to the first machine. When the first machine received the token, it sent out window sized packets(20 in my case) and passed the token to the second machine. After the second machine did the same processing, it passed to the third machine, and so on. Remember that this is the first stage and we will have two stages in total. The first stage is responsible for sending out packets. The second stage is

responsible for collecting aru information such that we know the next expected packet index to deliver.

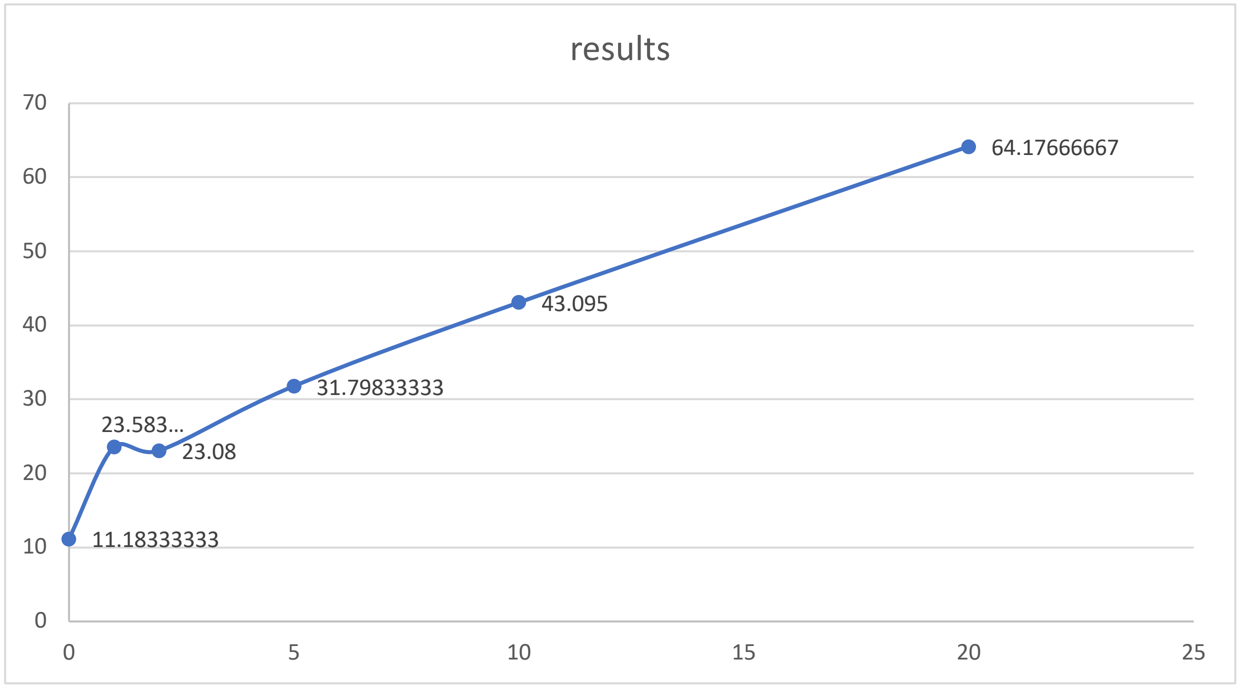
2. This is the second stage. Each machine will try to decrease the token.aru with their local arus such that we know for sure the next expected packet index for delivery. This is useful because if the token.aru is equal to token.seq, we will know that the current round is over and we can start sending new packets for the next window.
3. To process the received packets, we simply store all the received packets within the current window in a local 2D array. We need to check the array from the lowest window index to the highest window index to find out the local aru. This array is also important when updating the token.rtr for retransmission. When delivering, the corresponding array slot is set to null to be ready for the next window transmission.
4. The token needs to be retransmitted when missing. The timeout in my case is 10ms. When retransmitting, it is important to keep machines from processing duplicate token. If duplicate token is processed, inconsistency will happen. For example, the correct packet indices output for 3 machines and 20 window size should be 1,1,1,2,2,2,3,3,3…. However, when inconsistency happens, 21,1,1,2,22,2,3,3,3…may appear. This is because the entire window is corrupted such that our basic assumption that all the packets under transmission should be within the current window, is broken.

My original design is poor because even though it is using lamport algorithm to keep the agreed order, it still runs in round by round manner, which means I combined the worst parts of both free access protocol and token-based protocol. It is highly inefficient because we need acknowledgement for each packet. Even though I made a bit improvement to wrap many acknowledgements in a single packet, it is still generating too many packets because of the packet to packet communication. In order to maintain the transmission states, I also used a lot of complicated data structures including priority queue, hashset, hashmap, etc, which adds on more time cost. If 5 or less machines are running at the same time, it could finish transmission within few minutes. However, if more machines are running or higher loss rate is applied, the transmission almost halts.

Experiments:
The data is collected from the last finished machine. When the loss rate is increased, it becomes harder to make all the machines stop at the same time. When the loss rate is 20%, only one machine will finish and show the time cost, the other machines are still waiting for further packet transmission. In this case, I have to take the only one available data. At least it shows a trend.

loss rate	1st (s)	2nd(s)	3rd(s)	4th(s)	5th(s)	6th(s)	avg
0	11.38	11.08	10.86	11.12	11.33	11.33	11.1833333
1	23.28	23.4	23.87	23.87	23.85	23.23	23.5833333
2	22.8	22.84	22.7	23.89	23.02	23.23	23.08
5	31.86	31.77	31.69	30.98	32.53	31.96	31.7983333
10	42.51	43.67	43.48	43.6	42.81	42.5	43.095

20	64.76	63.76	65.73	59.54	66.27	65	64.1766667



It is clear that when the loss rate increased, the time cost for the transmission also increased. This result can be observed by the fact that all the machine can finish transmission almost at the same time when the loss rate is low, while could hardly synchronize when the loss rate is as high as 20%.
The token-based protocol is highly efficient because we are waiting for one packet to acknowledge 20 packets in my case. This is much more efficient than my original design where each packet is expected to be acknowledged by another packet, which generates a huge amount of packets and incur traffic congestion and high rate of packet loss.

Conclusion:
If the lamport algorithm is correctly implemented, it might be comparable to the token based one. However, token-based protocol is much easier to implement because there are less factors to control and less data structure to manipulate.