Distributed Systems Exercise 2 Design Document

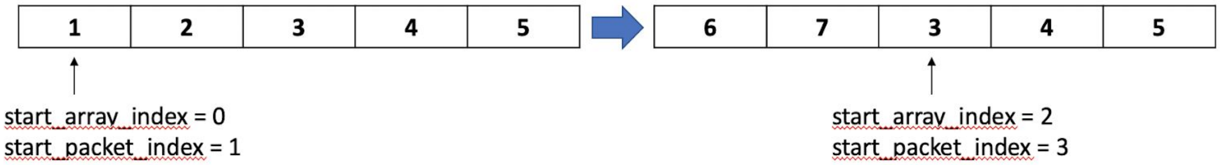
Minqi Ma (JHED: mma17), Jerry Chen (JHED: gchen41)

General Approach

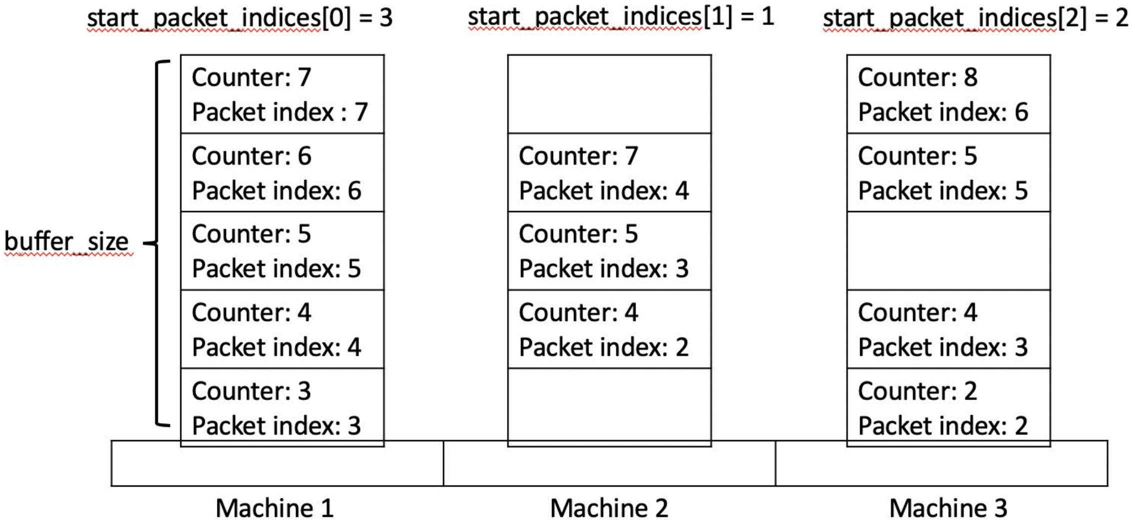
- Use Lamport timestamps to deliver packets in agreed and FIFO order
- Create a table to store received packets; deliver packets with the same counter at once; stop delivering if one packet is missing or out of order
- Send cumulative ACK packets to indicate the packet index **received** up to
- Send NACK packets in multicast; only the creator of the packet can respond
- Exit the program if every machine has the same last counter

Data Structures

- *struct packet* created_packets*: array of size CREATED_PACKETS_SIZE to store created packets
    - Note: to discard a delivered packet and create a new one (i.e sliding window), we wrap around and directly replace the old packet with the new one
    - For example: WINDOW_SIZE = 5, and we want to discard packet 1 and 2 and create packet 6 and 7



- *int* acks*: for machine i, how many packets from the current machine it has **received** up to
    - For example: machine_index = 1, acks = [1 2 3] means machine 1 has received up to packet 1 from machine 1; machine 2 has received up to packet 2 from machine 1; machine 2 has received up to packet 3 from machine 1.
- *struct packet** table*: table of size TABLE_SIZE = num_machine * buffer_size to store received data packets from other machines.
    - Packets are placed in the table according to its machine_index and packet_index
    - Each array use wrap-around technique (same as above) to avoid shifting entries
    - For example, num_machines = 3, buffer_size = 5. *table* can look like:



- *int* end_indices*: array to store last packet index for each machine
- *bool* finished*: for each machine i, if current machine has finished **delivering** its packets
- *int counter*: current counter to support Lamport timestamps

- *int last_delivered_counter*: the last counter that current machine has finished delivering
- *int* start_array_indices, int* start_packet_indices*: parameters stored to help wrap around arrays
- *bool ready_to_end*: if current machine has finished delivery and received all acks (explained in Algorithms part)
- int* last_counters: array to store last counters of each machine when they finished delivery. A machine only sends its last counter through a COUNTER packet when it's ready to end.
- struct timeval *timestamps: record most recent transmission time for each packet in *created_packets*

Message formats

struct packet:

- *unsigned int tag*: indicate the type of the packet
- *unsigned int counter*: counter of the packet
- *unsigned int machine_index*: index of the machine which created the packet
- *unsigned int packet_index*: packet index (or sequence) of the packet
- *unsigned int random_data*: a random integer between 1-1000000
- *int* payload*: unused data, or information about ack, nack or last counter (as specified below)

Types of packets:

- TAG_START: machines can start transmission
- TAG_DATA: data packet which has *counter*, *machine_index* and *random_data*. *payload* is not used.
- TAG_ACK: first <num_machines> integers in *payload* record, for each machine i, the number of packets from machine i that current machine has received up to
- TAG_NACK: first <num_machines> integers in *payload* record, for each machine i, packet index of the missing packet; -1 indicates no missing packet from machine i
    - For example, num_machines = 3, machine_index = 1, payload = [-1 3 5] means that machine 1 does not miss any packet from machine 1; machine 1 misses packet no.3 from machine 2; machine 1 misses packet no.5 from machine 3.
- TAG_END: *packet_index* contains the last packet index of the machine
- TAG_COUNTER: first <num_machines> integers in *payload* record the last counter for each machine. -1 indicates unknown.

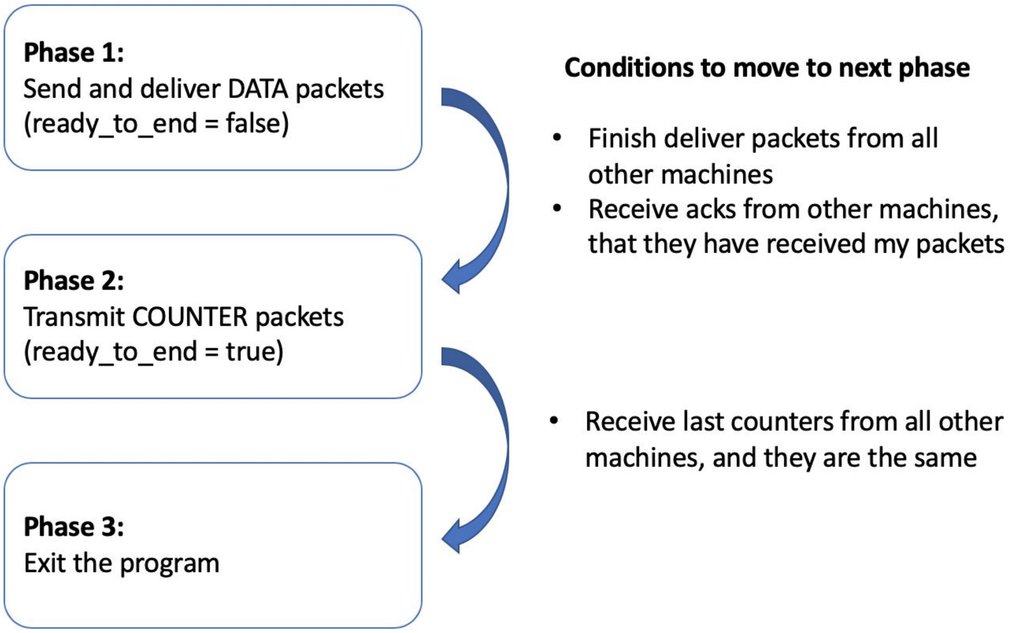
Algorithm

Lamport timestamps

- When creating a packet, increment counter and stamp the packet with counter, machine_index and packet_index
- Adopt the larger counter when receiving a DATA packet
- Deliver in lexicographical order of <counter><machine_index>

Termination logic

- Each machine goes through 3 phases (to satisfy 3 termination conditions):



Phase 1:
Send and deliver DATA packets
(ready_to_end = false)

Phase 2:
Transmit COUNTER packets
(ready_to_end = true)

Phase 3:
Exit the program

Conditions to move to next phase

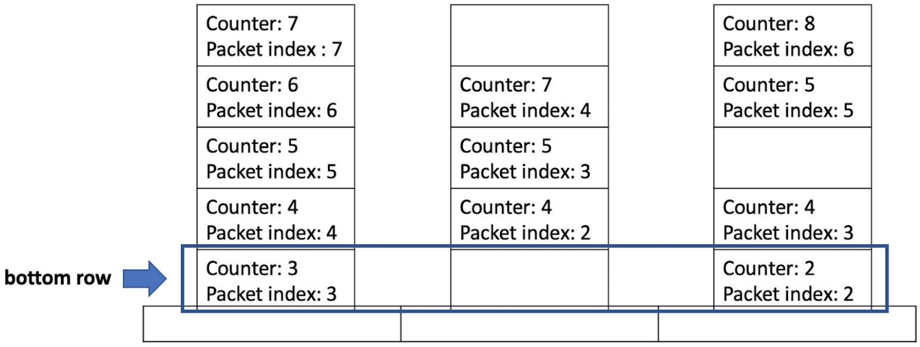
- Finish deliver packets from all other machines
- Receive acks from other machines, that they have received my packets

- Receive last counters from all other machines, and they are the same

- In Phase 1, every time after delivery or updates on *acks*, we check if the machine satisfies 2 conditions to move on to Phase 2.
- In Phase 2, every time after receiving COUNTER packet, we check if the machine satisfies the condition to exit the program.
- Note that *ready_to_end* distinguishes Phase 1 from Phase 2.

Protocol

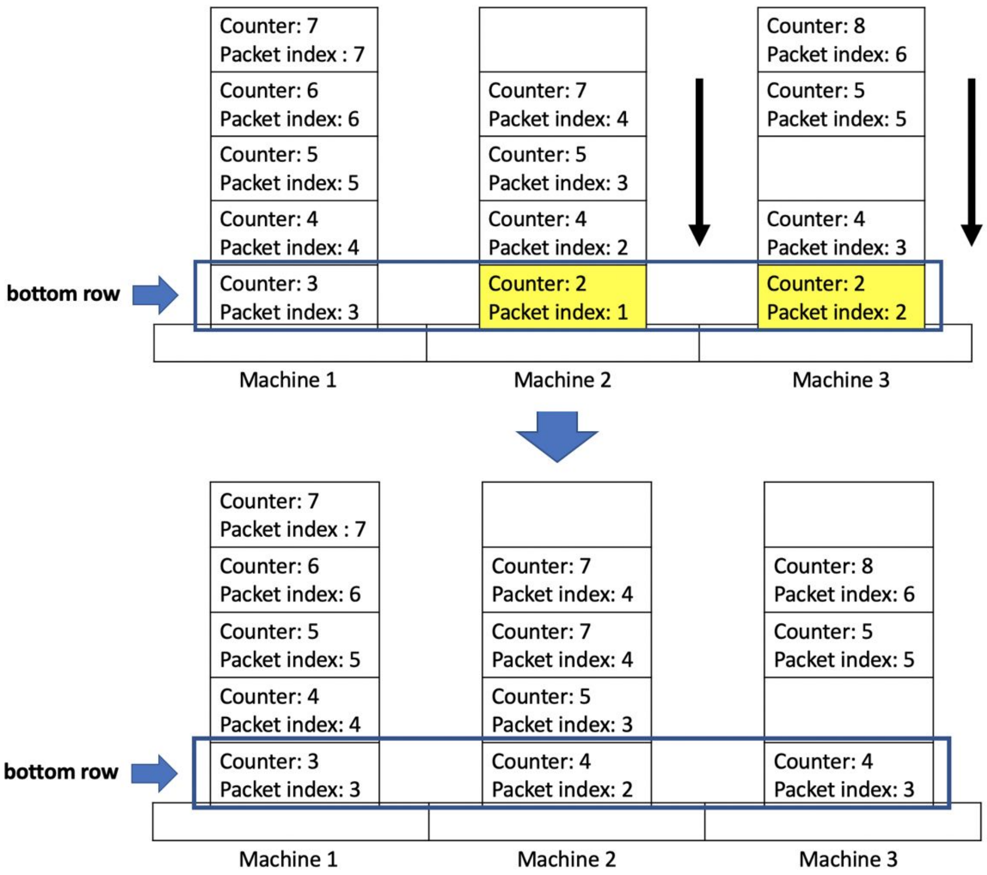
- Receive TAG_START packet:
    - Generate packets in *created_packets* array
    - Send a fraction of the window in multicast (to avoid overwhelming the network)
- Receive TAG_DATA packet:
    - Insert packet to the corresponding empty cell in the table
    - Adopt the larger counter
    - If received packet is out of order and there are gaps in the buffer, send NACK
    - Check if we can deliver if among unfinished machines, the bottom row in the table are full (i.e has received the next packet to deliver for each machine)
        - Update NACK array accordingly if any packet is missing or out of order in the buffer
        - If it can deliver, only deliver those with counter == last_delivered_counter + 1
    - Delivery will stop if the bottom row is not full; then send NACK packet
    - Send ACK for every ACK_GAP packets received
    - <u>Examples</u>: last_delivered_counter = 1



	Counter: 7 Packet index : 7		Counter: 8 Packet index: 6
	Counter: 6 Packet index: 6	Counter: 7 Packet index: 4	Counter: 5 Packet index: 5
	Counter: 5 Packet index: 5	Counter: 5 Packet index: 3	
	Counter: 4 Packet index: 4	Counter: 4 Packet index: 2	Counter: 4 Packet index: 3
bottom row	Counter: 3 Packet index: 3		Counter: 2 Packet index: 2
	Machine 1	Machine 2	Machine 3

*This is in fact *created_packets* array

        - CANNOT deliver because bottom row is empty; send NACK for packet no.1 of machine 2 and packet no.4 of machine 3



        - CAN deliver because bottom row is full. Can ONLY deliver highlighted packet since *counter* == *last_delivered_counter* + 1. After delivery, packet will be discarded and heads of array are slided. (can be interpreted as the rest of cells "fall" down)
    - If deliver packet with the last packet index, mark the machine as finished in *finished* array
- Receive TAG_ACK packet:
    - Update *acks* array from *payload* information. If created packets have been received by all machines, slide window and create new packets.
    - Record timestamp for transmission
- Receive TAG_NACK packet:
    - Only respond if it is requesting current machine's packet
    - Look up in *table* and *created_packets* array. Retransmit the packet if the interval between last transmission and now is larger than RETRANSMIT_INTERVAL
    - If the requested packet index exceeds last packet index for the machine, send END packet.
- Receive TAG_END packet:
    - Record last packet index for the machine in *end_packet_indices* array
    - Update *finished* array if necessary.
- Receive TAG_COUNTER packet:
    - Update *last_counters* array accordingly
    - Send its own last_counters array in COUNTER packet if *last_counters* array gets updated
    - If received all last_counters and are the same, send COUNTER packets for 5 times and exit.
        - Note: once one machine receives all last_counters, we would like to inform all machines to exit. To make sure that the packet won't be lost, we can send it for 5 times, as the highest loss rate is 20%.
- Timeout:

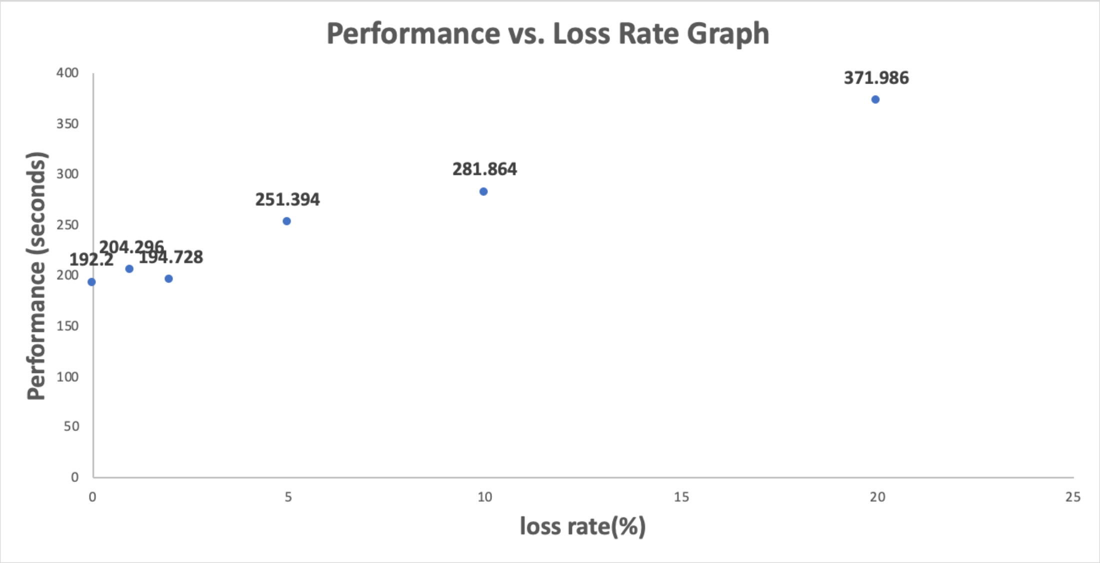
- In Phase 1:
    - If haven't finished delivery, resend most recently sent NACK packet and current ACK packet
      If have finished delivery, send END packet with last packet index
    - If min(*acks*) != *num_packets*, (i.e not all packets have delivered packets from the current machine), send the packet in the window with the highest packet index
- In Phase 2: send COUNTER packet

Changes made

1. Originally we only send NACK when the delivery stops. Now we first send NACK when the received packet is out of order in the buffer, since we know that there must be packets missing in between. It will save some time from sending NACK early.
2. Originally every machine will respond to NACK as long as they have the requested packet. Now only the owner of the requested packet will respond, so the network won't be overwhelmed with multiple responses for the same NACK.
3. We change TABLE_SIZE from a couple thousands packets to less than 500 packets.
4. Originally we respond to NACK everytime we receive it. Now we add an array of timestamps to record the most recent transmission time, to avoid multiple retransmission for the same packet in a short amount of time.

Performance

loss rate(%)	1	2	3	4	5	average performance (s)
0	117.29	215.52	153.97	231.87	242.35	192.2
1	249.16	147.17	239.69	237.88	147.58	204.296
2	209.21	139.04	133.5	156.87	335.02	194.728
5	173.28	289.23	327.85	317.9	148.71	251.394
10	250.37	272.95	241.65	341.14	303.21	281.864
20	566.65	418.62	498.29	513.72	453.09	490.074



Discussion

- Obviously, it takes more time to transmit with a higher loss rate, because packets will be discarded in the midway and it requires more NACK packets to get them back.
- Unfortunately our programs timed out at 100-second limit. There might be something wrong with the core design or the tuning parameters. We have spent a lot of time on both directions, but none of the changes have made a huge impact on the result. This is the best results we can have after multiple trials.
- We also notice that the result of the program varies a lot even with the same input. There might be some issue with our flow control, that it sometimes overflow the network and sometimes not, especially for ACK and NACK packets.
- We chose to use arrays over list, because we though O(1) access will be more efficient than O(n). We did not know how the usage of array and list will affect the performance.