Practical Exercise 2

Design Document: Token Ring Multicast Protocol

Chas Forelle

Token Ring Phases

<u>Setup Phase</u>

a. Wait for start_mcast signal (using recv)

b. Begin waiting for responses
    a. Multicast process id and IP address
    b. Process 1 waits until it has a full ID/IP list, then attaches to token and sends it. Wait for token to come back and then begin transfer phase sending state via generating first token.
    c. All other processes multicast their ID/IP on timeout until receive the token. Make sure they have all ID/IP via list on token and pass. Move to transfer phase receiving state.
    d. If a process times out and has seen the token, retransmit the token. Otherwise multicast id/ip info again.

<u>Transmit Phase</u>

a. *Receiving State*
    a. Read input as data packet type
    b. If data.seq > local.seq --> local.seq = data.seq
    c. If local.aru < data.seq --> add data to the circular array
    d. Increment local.aru up to next missing data
b. *Sending State*
    a. Read input as token packet type
        i. *If token.seq < local.seq it is an old token, ignore it and return to receiving state*
    b. If this process is process 1, increment the token.round
        i. *Used to determine termination of the ring*
    c. If we decremented token.aru last round, raise it
        i. *If saved_aru == token.aru then set token.aru == local.aru and saved_aru = -1*
    d. Else if local.aru is lower than token.aru, lower token.aru and retain value
        i. *Local.aru < Token.aru? token.aru = local.aru and saved_aru = local.aru*
    e. Update the retransmit list on token
        i. *For each rtr order, send if possible and remove from list if sent*

        ii. *If local.aru < local.seq then process is missing packets between [local.aru] and [local.seq]. Iterate thru the indices and add rtr order to the token for each*
        iii. *Update the token.rtr_len = sizeof(rtr list)*
    f. Send up to (20 - 40) new data packets multicast, where data.seq = token.seq++. Increment local.seq, local.aru and token.aru as appropriate. Store new data into array.
    g. Save the new token, store previous token.aru
        i. *Useful if this process has to retransmit the token, and for deciding when to write*
    h. Send the token
    i. Deliver (write to file) all data in array up to min(saved_token.aru, old_token.aru)
        i. *Saved_token is the most recent sent, old_token.aru was saved when we overwrote it in (f)*
        ii. *Do the write step last because it is expensive and the next process can begin processing instead of waiting for us to write to file*

c. *Timeout State*

    a. If this process hears nothing after a timeout (maybe 100ms) it can assume the token was lost and it needs to resend it.

<u>Termination Phase</u>

a. During sending state in Transmit Phase (process has token) the process can decide it is done
    a. *If the token.aru = token.seq = local.seq = local.aru and no rtr orders*
    b. *This condition is 'final' state. If (a) fails in any of the following steps, the ring is not complete and we need to restart termination phase.*
b. Once process decides it is done, it stores token.round and passes token
    a. *Process thinks it is finished, unsure if others are finished*
c. Process must see 'final' token with token.round = saved_token.round + 1, then pass token
    a. *Process knows it is finished and everyone else is finished, unsure if others know this*
d. Process must see 'final' token with token.round = saved_token.round + 2, pass token
    a. *Process knows that everyone knows they are finished and can exit safely*
    b. *Only process that initiates termination needs to see round+2, others can see round+1 and timeout safely*
e. Exit

Data Structures

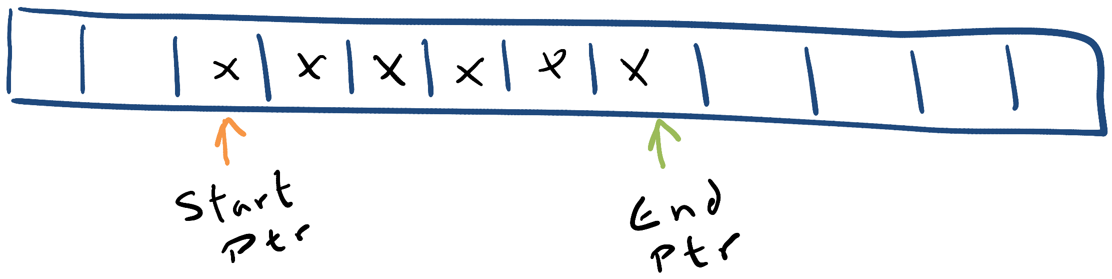
<u>Packet Types</u>

```
initializer{

      char type; //packet identifier (init=1, token=2, data=3)

      int src_p; //source process id
```

```
        int src_ip; //source process ip
    }
    token {
        char type;
        int src_p;
        int src_ip;
        int round; // process 1 stores current round, prevents dups
        int seq;
        int aru;
        int rtr_len; //number of retransmit requests in rtr
        int* rtr;
    }
    data {
        char type;
        int src_p;
        int seq;
        int rand; //random data that is delivered (written)
        char payload[1200]; //junk bytes of payload data
    }
```

<u>Local Data Storage</u>

A circular array will be used to store local packets until they are ready for delivery to the file.



The array is an array of pointers of some length (likely 1000). All active data is between the start_ptr and the end_ptr. Access of the queue begins at the start_ptr and steps through to the end_ptr. Adding elements puts them in front of the end_ptr and moves the end_ptr, or fills an empty index in between start_ptr and end_ptr. Discard if index is filled or data.seq > 1000 + local.seq, as it will overwrite stored data. Removing elements takes them from the start_ptr and then increments the start_ptr. The max size of the queue is length-1, so as to avoid zero-full collision (two cases when start_ptr = end_ptr, are we full or empty?). Moving the start and end ptrs will require a mod operation, in order to 'ring' back around to the start.

Storing data in the array will require a struct like {int seq, int src_id, int random_data}, in order to store the packet index, machine index and random data.

Conclusion

Our token ring does not successfully complete a transfer phase. There appears to be an off-by-one error (or multiple) that is preventing the aru from progressing past 0, and adding additional 'blank' data packets at number_of_packets + 1. Data is being successfully received and retransmitted, but as none of the processes have the packets_to_send+1 packets it gets stuck in an infinite loop trying to send data it doesn't have.

We made many changes to our design logic from above. Tokens carry a decremented id that 'locks' the token.aru from being incremented by anyone other than the process who decremented and locked it. Only another process can decrement it further, thereby re-locking the aru. This fixed some cases where the token could return back to a process and it would mistakenly think it still had the lock. Additionally, the retransmit list is updated slightly differently. A new retransmit array is created and populated, instead of modifying the current retransmit list. This simplified the logic required to maintain the list. The array queue originally had a start and end pointer, but in order to simply logic we changed the management to a single start_ptr and a length variable. The start_ptr held the absolute (un-modded) index in order to be easily compared to local aru and packet.seq.

Our program does not implement a termination phase because of the aforementioned off-by-one errors, but if we could have found and fixed them we would have been able to easily add the end conditions.

Because the processes cannot increment aru past 0, none of them are able to write off to file. The values remain in memory until the process is closed.