600.337 Distributed Systems Assignment 2

Yixiao Wu [ywu66@jhu.edu](mailto:ywu66@jhu.edu)

Sihao Lu [slu21@jhu.edu](mailto:slu21@jhu.edu)

We are implementing the agreed ordered multicast protocol using Single Ring protocol thanks to the article [*The* ***Totem Single****-****Ring*** *Ordering and Membership Protocol*](http://www.cs.jhu.edu/~yairamir/tocs.ps)by Prof. Amir.

1. Structure
   1. Start\_mcast

Start\_mcast contains only one packet structure with the following field:

* *type*: START

Briefly speaking it just send out a special message indicating that all mcasts should start transmitting messages.

* 1. Mcast

A regular packet that contains regular messages is structured as follows:

* *type*: REGULAR
* *sender\_id*: The machine number of sender of the packet
* *seq*: The message sequence number
* *contents*:
  + first 4 bytes as a random integer
  + the rest 1200 arbitrary payload bytes.

A preparation packet is used to let every machine know its immediate neighbor in its “next position” in order to know who to send tokens to in the future. It is structured as follows:

* *type*: PREPARATION
* *sender\_id*: The machine number of sender of the packet
* *seq*: Ignore
* *contents*:
  + *rcved*:A boolean indicating if the machine has received the IP address of the next machine
  + Sender’s IP address.

The token is structured as follows:

* *type*: TOKEN
* *sender\_id*: The machine number of the sender
* *seq*: Here it represents token sequence number
* *contents*:
  + *msg\_seq*: The highest sequence number of the message that has been multicasted.
  + *aru*: A low watermark indicating that all machines have received messages with indices until this number
  + *aru\_id*: The index of machine that set aru
  + *nacks*: A list of negative acknowledgemen`ts (all the sequence numbers of missing messages) to request retransmission.

Mcast needs to maintain records of messages/tokens so it must require a set of local variables to store the status information. They are shown below:

* *last\_token\_seq*: The sequence number of last token forwarded. Used to prevent token duplication.
* *my\_aru*: The sequence number of the message such that the machine has received all messages with sequence number less than or equal to this number.
* *send\_queue*: A queue of messages from the local machine that will be sent.
* *receive\_queue*: A queue of messages received that are waiting to be processed.

The send\_queue and receive\_queue both has a limited size equal to a constant WINDOW\_SIZE. Upon the transmission we utilize the sliding window technique as we discussed in the class.

1. Protocol
   1. Pre-preparation stage: every mcast client is setup with its machine index and the knowledge of total number of machines. Start\_mcast initiates and send a lossless message to every mcast, thus the protocol starts and goes to preparation stage.
   2. Preparation stage: every mcast **keeps sending** a PREPARATION packet containing its machine index, IP address and *rcved =* 0 infrequently (in order not to blow up the Internet) to everyone else using the packet structure as previously described in “Structures” section. If it receives the IP from next machine, it changes *rcved* to 1 and send the new version of PREPARATION packet. It does not stop sending until it reaches a timeout (which will leads the mcast to fail) or it receives a TOKEN packet from its neighbor in its previous position in the ring (e.g. machine 5 receives message from machine 4). The mcast may leave this stage due to any of these three reasons:
      1. If it stops due to timeout, it aborts. Otherwise we have situation ii and iii.
      2. Machine 0: It maintains a vector table of every other machines’ *rcved* field in their packet received. If all of them are 1, go to transmission stage.
      3. Others: It goes to Transmission stage when it receives a TOKEN from last machine.
   3. Transmission stage: In the beginning, machine 0 generates a token with token sequence number = 0, message aru = 0, message sequence = 0, aru\_id = 0, nacks = []. Then it sends the messages and passes the token to its next neighbor according to the steps below.
      1. Only the machine with token can send messages.
      2. Upon receiving the token, if the sequence number of token is less or equal to the local *last\_token\_seq*, meaning this is a redundant token, it discards it.
      3. Upon receiving the non-duplicate token, the machine updates *last\_token\_seq* to the token’s sequence number and increase the token seq by 1. Then the machine updates the token’s *aru* to *my\_aru* if it is larger than *my\_aru,* then it sets the *aru\_id* in the token to its own machine index, indicating this machine changes the aru. If the token’s *aru\_id* is the same as machine index, meaning no other machines updated the aru in this round, the current machine updates its *my\_aru* to *aru* in the token and wipes *aru\_id* to null.
      4. When it’s a machine’s turn to send, so it first check the *nacks* field from the and send the messages with sequence numbers equal to the numbers in *nacks* first while removing these from *nacks*, then it sends the messages with sequence *number* from *aru* to *aru* + WINDOW\_SIZE. Every time before it sends a message, it increases the token’s message sequence number *msg\_seq* and set the outgoing message’s *seq* to that number.
      5. When the token’s *msg*\_*seq* is larger than the local *my\_aru*, which means the machine missed messages, it adds all numbers between *my\_aru* + 1 to *msg\_seq – 1* to the *nacks* field.
      6. It sends the token to next neighbor and waits for a constant amount of time called token retransmission timeout. Any reception of messages or tokens cancels the timeout. If it doesn’t hear from its previous neighbor after the timeout, which means the token it sent has been lost, it retransmits the token and resets the timeout.
   4. End-of-Transmission stage:

A machine remembers the msg\_seq and aru of the token on previous round. If after one rotation the new token it receives keeps the same msg\_seq and aru, there should be two situations:

* + 1. The token has no *nacks*: it exits as it has finished transmission.
    2. The token has *nacks*: it sends the packets mentioned in the *nacks* and keep passing the token.