CS425 MP1

Name: Funing Xu NetID:fxu12

Step 2 Implementation:

Delay time:
Delay time is specified in the first line of the configuration file. Channel first reads the min_dealy and max_delay value from the configuration file and convert it from milliseconds to seconds. The actual delay time is a randomly generated number by calling “random.uniform”, whose range is between min_dealy to max_delay.

Unicast:
When the message is handed from process to its channel, a threading containing the actual unicast function will be started after certain amount of time passed, where the time is defined as delay time above. The delay functionality is achieved by calling “threading.Timer”. Inside the actual unicast function, a UDP socket will be created and it will send the user input message to the destination process.

Since we create a new thread for each unicast with different delay time, each thread will be started at different time and the requirement of non-FIFO delivery is met.

Unicast receive:
A process first creates a socket and bind to the specified IP address and port, then waits for individual messages. Every time a message is received, it is handed to the channel. Since the delay functionality is implemented during the unicast phase and there is no ordering requirement for this step, the channel would deliver the message to its process.

Since the existence of the delay, the order of the message received by the receiver process would be different from the order when those message are sent.

Step 3 Implementation:
Causal ordering and total ordering are implemented in “CausalOrderChannel” and “TotalOrderChannel” classes. Both of the classes inherit from the delayed channel as described above.

Causal ordering:

Algorithm for group member p_i $(i = 1, 2\ldots, N)$

On initialization

$V_i^g[j] := 0$ $(j = 1, 2\ldots, N)$;

To CO-multicast message m to group g

$V_i^g[i] := V_i^g[i] + 1$;

B-multicast$(g, <V_i^g, m>)$;

On B-deliver$(<V_j^g, m>)$ *from* p_j, *with* $g = group(m)$

place $<V_j^g, m>$ in hold-back queue;

wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \leq V_i^g[k]$ $(k \neq j)$;

CO-deliver m; // after removing it from the hold-back queue

$V_i^g[j] := V_i^g[j] + 1$;

On initialization:

The implementation of causal ordering channel follows the above algorithm as shown in the slides. Array represented vector timestamp and hold back queue are instantiated in the channel constructor. The length of the vector timestamp array is the same as number of processes specified in the configuration file and the value of each element in the array is set to zero as default. Hold back queue is instantiated as an empty array.

CO-multicast:

Each time we call the multicast function, we first want to update our vector timestamp. Since it is possible that the multicast function are called at the same time, we add a lock to protect the access to our vector clock. Here we are using “with” statement with a lock every time we want to update the vector timestamp. For each multicast action, we increment one to the entry corresponding to the sender. For example, suppose we have 4 processes, and the sender is process 1, then we would update by executing “vector_timestamp[0] += 1”. After that, we send the message with the information about vector timestamp to all the processes belonging to the group.

On B-deliver:

If the message is sent from itself, we would deliver the message without checking. Otherwise, we first check the vector timestamp from the message by calling our “check_order

“ function. If the return value is false, we would push the message to our hold back queue. If the return value is true, we would deliver the message to the process.

Inside the checking function, suppose the sender is **j** and the receiver is **i**. We first check if i_vector_timstamp[i] + 1 equals to j_vector_timstamp[i], if it is not, we would return false. Then we want to check every other indexes to see if it satisfies the condition “j_vector_timstamp[k] <= i_vector_timstamp[k]”. If all the indexes of the vector timestamp satisfy the condition, we would return true, otherwise false.

If the vector stamp we are getting is valid, the receiver process would first update its own vector timestamp by increment one to the sender entry. Then, the message will be delivered to the process.

After that, we want to check our hold_back queue to see if there is any message with valid vector stamp since we just update the receiver vector timestamp. We do the same check as described above to every message from the queue. If there exists such a message with valid vector timestamp, the channel will deliver it to the process and update its vector timestamp correspondingly.

Total ordering:

1. Algorithm for group member p

On initialization: $r_g := 0$;

To TO-multicast message m to group g
  B-multicast($g \cup \{sequencer(g)\}$, <*m, i*>);

On B-deliver(<*m, i*>) *with* $g = group(m)$
  Place <*m, i*> in hold-back queue;

On B-deliver(m_{order} = <“order”, *i, S*>) *with* $g = group(m_{order})$
  wait until <*m, i*> in hold-back queue and $S = r_g$;
  TO-deliver m; // (after deleting it from the hold-back queue)
  $r_g = S + 1$;

2. Algorithm for sequencer of g

On initialization: $s_g := 0$;

On B-deliver(<*m, i*>) *with* $g = group(m)$
  B-multicast(*g*, <“order”, *i*, s_g>);
  $s_g := s_g + 1$;

The implementation of the total-ordering follows the sequencer approach. The sequencer is one of the communication processes. Process 1 is set as the sequencer by default.

On initialization for group member p:
A sequence number r = 0 is initialized. Two empty hold back queues are initialized. One of the queue is for message passed from other process, another one is from the sequencer. The reason why we have a queue for sequencer message here is that it is possible the order message sent from sequencer could arrive earlier than the actual corresponding message sent from processes.

On initialization for sequencer:
Besides the member variables mentioned above for a regular process, the sequencer initializes a consecutive increasing sequence number s starting from 0.

Message:
There are two types of message used in this algorithm. First is the regular multicast message, the only difference is that the message contains a ID as unique identifier. The other one is the order message sent from a sequencer. This message contains a ID and a sequence number.

TO-multicast:
Every time we multicast a message, we first generate a random number ranging from 1 to MAX_INTEGER used as a unique identifier and bind this number to our message. Then we just B-multicast the message to all the processes in the group including the sequencer.

Process on B-deliver with process message:
After we receive a message from other processes, we first put the message into our hold back queue. Then we want to check our sequence queue to see if we already received a order message first. If there exists a order message with sequence number equals to r, and a message with same ID also appears in the hold back queue, then we want to deliver that message to the process. After that, we would like update the receive sequence number r by increment one.

Process on B-deliver with sequencer order message:
When a order message sent from sequencer is received, we first check our hold back queue to see if the queue contains the message with corresponding ID, then we want to compare the sequence number received with our local sequence number. If two numbers equal, we can then deliver the message to the process and update the local sequence number. After that, since the sequence number is updated, we can double check our sequence number queue to see if there is a match.

Sequencer on B-deliver with sequencer message:

If a process is also a sequencer, after a multicast message is received, it will multicast a order message with the id retrieved from the received multicast message and its sender sequence number. After the order message is multicasted, the sequencer will update its sender sequence number.

How to run:

1.no ordering:
open up a terminal, type ‘python process.py id regular’, ‘id’ represents process id, you can change the number as long as it is specified in the configuration file.
For example, to start process 1, just type ‘python process.py 1 regular’.

To send a unicast message to a specific process, you can type “send process_id message”, for example, you can type “send 2 hello” which sends a string “hello” to process2.

Note: the no ordering implementation doesn’t multicast.

2. causal ordering
type ‘python process.py 1 causal’ to start a process 1 in causal ordering.
To send multicast message, just type “msend message”, where message is the actual message you want to send.
For example, you can type “msend hello” to send hello to all the processes in the group.

3. total ordering
type ‘python process.py 1 total’ to start a process 1 in total ordering. The send command is the same as causal ordering.