HW3 Solutions: CS425 FA20

1. (Solution and Grading by: <Yigong>.)
   Messages may take arbitrarily long so in the last round when pi gets a message from pk
   but pj does not, it does not mean that a third process pk failed (could mean pk to pj
   message was delayed much longer).
   (Other reasonable answers also acceptable)

2. (Solution and Grading by: <Binyao>.)

3. (Solution by: <Binyao>. Grading by: <Yuyang Liu>)

4. (Solution and Grading by: <Ishani>.)
(i) You don’t quite understand how, in Lemma 3, Case II, one can have p’=p, especially since the set D is obtained from events in set C by not applying event e=(p,m)! Can you explain this discrepancy?

In case II, it is ok to have p=p' because a process p may be delayed long enough or a message e=(p,m) may be delayed long enough (these are possible in an asynchronous system).

(ii) Why in the proof of Lemma 3, Case II does C definitely have a deciding run, i.e., schedule S? What if C never decides? Does the proof not hold then?

We assume C has a deciding run because we are assuming that whatever consensus algorithm we are provided with, reaches consensus in a finite number of steps.

(iii) It appears that Lemma 3’s proof will hold no matter which event e is selected for the argument (e applicable on configuration C). Is this true, or does he need to have some special characteristics?

The conditions for e and e’ are stated at the start of the proof. D0=C0 folll. by e=(p,m) 
D1=C1 foll. by e=(p,m) 
And C1 = C0 followed by some event e’=(p’,m’)
Any e, e’ satisfying these will suffice.

5. (Solution and Grading by: <Atul>.)
Process states:
- NHX: S(f)
- Earth: R(c)
- Moon: S(b)

Channel states
- Moon -> NHX: {}
- Earth -> NHX: {}
- NHX -> Earth: {}
- Moon -> Earth: {}
- NHX -> Moon: {f}
- Earth -> Moon: {d}

6. (Solution by: <Ruiyang>, Grading by: <Camille Zhang>.)
The midpoint of the system is $M = 2^{\lfloor m/2 \rfloor} - 1$

a) A simplified version of the algorithm is as follows: route to midpoint M, then have that midpoint talk to its successor and predecessor, and have them decide who is the closest to the midpoint. Spread this information to everyone.
b) Safety: At most 2 processes are neighbors of the midpoint, and they get to decide who is the closest. At most one is elected. Liveness: Since there are no failures and messages are eventually delivered, all Chord messages are eventually routed, one leader is elected, and everyone knows about it.

c) Completion: O(N), messages: O(N). (Explanation: O(1) Chord routing messages to elect a leader, O(N) Chord routing or regular messages to inform everyone of the new leader.)

d) If there are failures, and finger tables are inconsistent, Chord routing may not be correct, and the two nodes deciding may not be the actual successor and predecessor of midpoint. Safety might be violated. As messages may loop around forever, Liveness may be violated.

7. (Solution and grading by: <Ruiyang>)

Sequencer: keep a global sequence number S and a vector of Pi’s for which Pi(j) tracks the sequence number for Process j.

For Each Process P(i): keep a local sequence number S_local and Pi

Algorithm:
When Process i wants to send a multicast M, it will increase Pi = Pi+1 and then send out <M, new Pi> to the group and sequencer.

When Process i receives a message M from peers, it will buffer the message until it receives <M, S'> from the sequencer (or it has already received it). It will deliver this message when S_local = S'-1 and update S_local to S'.

When a sequencer receives a multicast <M, Pi> from process j:
If the last message's sequence number Pi(j) == Pi-1, it will increment S=S+1 and Pi(j)=Pi, then send out <M, new S> to the group.
Otherwise: it will buffer message <M, Pi> until Pi(j) = Pi-1.

The total order is provided by the sequencer and FIFO is provided by each process’s Pi number.

8. (Solution by: <Xin>, Grading by: <Xin, Haorong Sun>)
Assumptions:

- We have n processes in the system p1, ... , pn with pi_id < p(i+1)_id for all 1 <= i <= n, which collectively elect 1 <= k <= n leaders.
- Processes are fail-stop.
- The algorithm doesn’t satisfy safety when there are less than k non-faulty processes (there would be null entries in the leader list).

Election Algorithm:
Initially there are no failures. The k leaders are p1, ... , pk and all processes recognize this initial state.

When a process pi detects the failure of a leader pf:
  a) If pi itself is a leader, it constructs the following Election message: <elect: null, remove: pf_id>.
  b) Otherwise pi is not a leader, and it constructs the following Election message: <elect: pi_id, remove: pf_id>.
  c) pi sends this Election message to its ring predecessors.

When a process pi receives an Election message <elect: elect_id, remove: fail_id>:
  a) If elect_id = pi_id, pi has been elected. Forward the following Elected message: <elected: pi_id, remove: fail_id> to ring predecessors.
  b) If (elect_id is null OR pi_id < elect_id) AND pi itself is not a leader: replace elect_id with pi_id.
  c) If pi has not forwarded this Election message (identified by fail_id) in the recent past, forward it to ring predecessors.

When a process pi receives an Elected message <elected: elected_id, remove: fail_id>:
  a) Replace fail_id with elected_id in its local leaders list.
  b) If pi_id != elected_id, forward the Elected message to ring predecessors.

If a process times out waiting for an Elected message after receiving an Election message, it re-initializes the election.

Note:
  a) As in the original token ring algorithm, we can handle multiple initiators by including initiator’s id in the Election message. The details are not included in the above algorithm for simplicity.
  b) There are at most k messages at any time in the system, since each detected leader failure induces one Election/Elected message, and there are at most k failed leaders.

Safety: Initially the leaders are the k processes with the lowest ids. Then if any leader fails, it is replaced by the process with the lowest id among non-faulty, non-leader processes (if exist).

Liveness: An election always terminates either because a new leader is elected or the election times out. For each non-faulty process, its leader list entries are never null (except when there are less than k non-faulty processes).

9. (Solution and Grading by: <Bhavana>.)
   Consider the following arrangement of processes. Here, the voting set of a process pi consists of processes in the same row and column as pi.
Consider the following scenario:
1. Nobody is in the CS and p9 wants to enter. p9 sends a request to \{p3, p6, p9, p7, p8\}. Since nobody is in the CS, all processes in the voting set send a reply and p9 enters the CS.
2. p3 wants to enter the CS and sends a request to its voting set \{p1, p2, p3, p6, p9\}. Since, p1 and p2 haven’t already voted, they send a reply to p3. However, p3, p6 and p9 enqueue p3’s request. At this time, p3 is waiting for a reply from p3, p6 and p9.
3. p6 wants to enter the CS and sends a request to its voting set \{p4, p5, p3, p6, p9\}. Since, p4 and p5 haven’t already voted, they send a reply to p6. However, p3, p6 and p9 enqueue p6’s request. At this time, p6 is waiting for a reply from p3, p6 and p9.
4. p9 exits from the CS and sends a release message to its voting set \{p3, p6, p9, p7, p8\}. At this time, the queues of p3, p6 and p9 contain p3 and p6’s request. According to the updated Maekawa rules, p3, p6 and p9 multicast a reply message to ALL waiting processes in their queue. Following this, both p3 and p6 will receive all pending replies and enter the CS. This violates the safety clause as multiple processes have entered the CS at the same time.

Gist: You can have two processes pi and pj waiting for a reply from a process pk which is the intersection of pi and pj’s voting sets (because pk has already voted for ph, say). Now, when ph sends a release message to pk, according to the updated rules of Maekawa algorithm, pk will send a reply to all waiting processes in its queue (pi and pj). This will allow both pi and pj to enter the CS which is a violation of the safety clause.

10. (Solution and grading by: <Zhanghao>)
   a. Organize N processes in a virtual ring. There are a total of k token messages per file on the ring. A process can read a file as long as it acquires any one of the tokens. A process can write a file if and only if it acquires all k tokens. A process retains acquired tokens until it exits the critical section.
   b. Safety is guaranteed because the algorithm ensures the following properties:
      i. At most 1 process may obtain write access to the file simultaneously: as all k tokens have to be acquired by a process to obtain write access to the file.
ii. At most k processes may obtain read access to the file simultaneously: as each read needs 1 token, there are a total of k tokens, there can be no more than k simultaneous readers.

iii. If any process has write access to F, no other process should be able to read it: as all k tokens have been acquired by the writer, no other process can acquire a token to obtain read access.

iv. If any process has read access to F, no other process should be able to write it: if any of the token is acquired by a reader, no other process can acquire all k tokens to obtain write access.

c. The algorithm may livelock. For example, two processes that want to write may each hold k/2 tokens, preventing each other and other waiting readers/writers from progressing. This can be addressed in at least two ways (other ways possible):

   i. Tie breaking methods: If two processes are competing to acquire all tokens, then we use a random timeout on each of the processes to backoff (after they fail to acquire). Then, the processes that would like to write can try acquiring all k tokens again.

   ii. Writers, after acquiring the first token, start a timer with an associated timeout, and if they cannot acquire all k tokens by the timeout, they release all acquired tokens and start from the beginning.

   These two techniques can be combined with each other.

d. The best case occurs when the k waiting readers are immediate successors of the writer. The worst case occurs when the k waiting readers are immediate predecessors of the writer. As per clause iii), it suffices to calculate the total number of token transfers for the synchronization delay.

   i. Best case: the total number of token transfers = 1 + 2 + … + k = (1 + k) * k / 2.

   ii. Worst case: the total number of token transfers = (N - 1) + (N - 2) + … + (N - k) = N * k - (1 + k) * k / 2.

e. This is your stage :)