

# CS138 Homework Assignment 3 Solutions

Spring 2009

1. *Slide XV-12 describes what happens both when a replica manager sends a gossip message and when it receives a gossip message. However, the last part of the pseudo code for receiving a gossip message (given below) is wrong (it might run forever). Please supply a corrected version.*

- *while there exists request  $r$  in  $rm_b.log$  such that*
  - $r.u.prev \leq rm_b.val.ts$* 
    - *update val (by applying  $r.u.op$ )*
    - *$rm_b.val.ts = merge(rm_b.val.ts, r.TS)$*

The reason that the receive might never terminate is that, as written, it could repeatedly process the same update request  $r$  ( $r.u.prev \leq rm_b.val.ts$ ). The updating of the local TS will not cause  $rm_b.val.ts$  to exceed the “previous” timestamp of  $r.u$ . Therefore, it is also necessary to mark each request as processed in the log, to prevent this kind of loop.

```
while there exists request  $r$  in  $rm_b.log$  such that  $r.u.prev \leq rm_b.val.ts$  &&  $r.processed == false$ 
```

```
     $r.processed := true$ 
```

```
    update val (by applying  $r.u.op$ )
```

```
     $rm_b.val.ts = merge(rm_b.val.ts, r.TS)$ 
```

2. *Lecture XIV discusses virtual synchrony. Slide XIV-14 discusses the use of “flush” messages as preparatory to installing a new view.*

- a. *Why is it important that a process forward all unstable multicast messages in its cache before it sends a flush message?*

Once a process receives flush messages from all other processes in the current view, it is assured that it has received all cbcsts from the previous view and thus any further ones it receives are duplicates. So if a process sends a flush message then forwards unstable cbcsts, the forwarded cbcsts might be discarded (and ignored).

- b. *In slide XIV-15, could  $P_3$  have sent flush messages before receiving cbcst1?*

Yes. It had no unstable cbcsts in its cache, and thus was free to send a flush.

- c. *In slide XIV-18, suppose  $P_2$  had started to do a cbcst and had sent it to  $P_1$  (only), but then both  $P_1$  and  $P_2$  crashed at the same time. Note that the definition of views allows there to be a difference of just one process in the memberships of successive views. How would this situation be modeled? What happens to the message that  $P_2$  attempted to multicast? (Note that there is a typo in the notes for this slide: the third sentence should state that  $P_3$  notices that it has an unstable multicast.)*

This would be modeled as two successive view changes, with no messages exchanged between them. Flushes for the second view would indicate that both views could be installed. The message that  $P_2$  attempted to multicast is completely lost.

3. *The bully algorithm is an election algorithm that was discussed in class: each participant is assigned a unique identifier; the currently running process with the highest identifier is the coordinator — when a process resumes execution after crashing, it starts an election We assume*

*that all running processes can communicate with one another via multicast, and the total number of processes (running and failed combined) is known. When a process fails, it fails cleanly and comes back up (with the same identifier it had earlier) in a finite amount of time.*

- a. *We'd like to design a variant of this algorithm, which we'll call the noblesse oblige algorithm: the current coordinator continues to be the coordinator, despite recovery of other failed processes, until it fails. Please describe the details of the algorithm.*

The coordinator periodically sends "I'm alive" messages to all the other participants. When participants don't receive such a message after a certain period of time, they multicast their IDs to all the other participants. The one with the highest ID becomes the new coordinator. When a participant comes back up after failing, it contacts at least one other participant to find out who the current coordinator is.

- b. *Suppose that it's possible for the network itself to fail, so that the collection of processes might be partitioned into any number of non-intercommunicating pieces. It's important that at most one of the pieces elects a coordinator. Furthermore, if at all possible, there should be exactly one piece with a coordinator. Explain how this could be done. (It may require some modifications to the election algorithm. What is required of a piece so that it may elect a coordinator?)*

Whichever piece has the coordinator must have a majority (greater than half) of the participants. Thus if the network partitions itself and the piece containing the coordinator does not have a majority, then the coordinator must step down. One approach is to use a variation of the above noblesse oblige algorithm. However, the "I'm alive" messages delineate finite terms of office for the coordinator and are now more properly called "I'm running for reelection" messages. So, coordinators are elected for finite terms. At the end of each term a new election is held. To win the election, a participant must both have the highest ID and must receive the multicasts of half the total number of possible participants (not counting itself).

4. *Explain why serial equivalence requires that once a transaction has released a lock, it may obtain no more locks, even if the transaction is guaranteed not to abort. Your answer should be fairly general — it should not depend on particular examples.*

Suppose transaction T1 locks object A, releases it, then locks object B. There could well be another transaction, T2, that first modifies object A, then modifies object B. Suppose all of T2 occurs after T1 has unlocked A but before it locks B. We now attempt to find a serially equivalent execution of the two transactions. Since T2 modified A, in the serially equivalent version T1 must come before T2: if T1 had merely read A, then it must get the value of A before it was modified by T2. If T1 itself modified A, then, since the final version of A is whatever T2 set it to, T2 must modify it last.

But, since T2 modifies B, in the serially equivalent version T2 must come before T1: if T1 merely reads B, then it must get the value set by T2. If it modifies B, T1's value must be set last.

Since we get contradictory requirements for the serially equivalent version, our conclusion is that no serially equivalent version is possible.