Computer Science 2221 **Homework 4 Answers**

Behdad Esfahbod **Student # 993505827**

---

1. (a) The proof of Lemma 2 changes. The rest of the proof (two other lemmas) hold. The lemma is that there is an initial bivalent configuration. We prove it:

   Suppose by way of contradiction that every initial configuration is univalent. From the Nontriviality we know that there is an initial configuration that correct processes decide 0, and there is another that processes decide 1. Clearly using these and flipping one initial value at a time, we can find two different configurations, one 0-valent, one 1-valent, and they differ in just one initial value, for process p. We can easily reach a contradiction by crashing p at the beginning of round 1, before it sends out any message.

   (b) i. Here is the algorithm:
   - Each process has a local variable m. Initially $m = 0$.
   - Any process in round 1:
     Send 1 to everyone
     Receive round 1 messages
     If received exactly $n - t$ messages all equal to 1, set $m = 1$
   - Any crusader in round 2:
     Send m to everyone
     Receive round 2 messages
     If received exactly $n - t$ messages all equal to 1, decide 1
     Otherwise decide 0

   The correctness:
   Termination is obviously satisfied.
   Agreement and Nontriviality: If exactly t processes crash before they send any message to a correct process, then it is easy to check that any correct process decides 1. Otherwise, there is a correct process that receives more than $n - t$ messages in round 1 and sets it's m to 0, and sends 0 to everyone in round 2 and every correct process decides 0. So both agreement and nontriviality hold.

   ii. If $t > 1$, No. The case that at most one process crashes in each round is a special case (subset) of this case. By theorem we know that it cannot be solved in less than $t + 1 > 2$ rounds. If $t = 1$, we already have an algorithm to solve it in 2 rounds.

2. (a) i. Here is the algorithm:
   - Each process has a local variable m. Initially $m =$SF.
   - Sender in beginning of round 1: Sends message to broadcast to everyone.
   - Any crusader in round 1 (including the sender after broadcasting the message):
     If received message m' from sender, sets $m = m'$.
   - Any crusader in round 2:
     Send m to everyone
     Receive round 2 messages
     If received at least $t + 1$ messages different from m, deliver SF
     Else if received at least $t + 1$ messages equal to m, deliver m
     Otherwise deliver SF

The correctness:
Termination is obviously satisfied.
Validity: If sender is correct. Every correct process i receives the sender's message m in round 1, and then the same message from every correct process j in round 2, unless a failure has been occured for j (crashed, or malicious, or not received sender's message in round 1, or couldn't send message to i in round 2). But there are at most t failures, so i would receive the correct message from at least $n-2t \geq t+1$ correct processes. Then i would deliver the correct message in round 2 if it has not delivered SF in first If condition. As the sender is correct, no correct process can send i any message other than m, so i cannot reveice $t+1$ messages different from m. So i cannot deliver SF in first If condition.
Weak agreement: Assume by way of contradiction that two correct processes i and j deliver two different non-SF messages m and m' respectively. There are still $n-t-2 \geq 2t-1$ other correct processes. From these $\geq 2t-1$, at most $t-1$ report something other than m to i, because otherwise counting them and j, means that i should deliver SF in the first If condition. By symmetry there are at most $t-1$ that report something other than m' to j. But as $2t-1 \geq 2(t-1)$, means that there is at least one correct process that reports m to i in round 2, and m' to j, which is a contradiction.

ii. I assume that $t \geq 1$.
Consider a scenario A that sender is correct and sends everyone message m, then every correct process should deliver m in the end of round 1.
Consider a scenario B that sender is correct and sends everyone message m', which $m' \neq m$, then every correct process should deliver m' in the end of round 1.
Now consider a scenario C that sender is malicious and for some group of at $t+1$ processes, acts like the message to broadcast is m, and for the other that are still at list $t+1$, acts like the message to broadcast is m'.
As there is only one round, any message sent by any process i cannot be based on any message received by i. Then message sent by processes other than sender cannot be based on the message received by any process from sender. Then we can easily see that correct processes that received m from sender in scenario C should behave same as in scenario A and deliver m, while the other ones behave like in scenario B and deliver m'. This violates Weak Agreement.

iii. Assume by way of contradiction that there is an algorithm that solves the problem. Divide processes in three groups 1, 2, 3, let the sender be in group 1.
Scenario A: Sender broadcasts message m, and all processes in group 2 are malicious and act to processes in group 3 so that if the message to broadcast is m'. By validity processes in group 3 should deliver m.
Scenario B: Sender broadcasts message m', and all processes in group 3 are malicious and act to processes in group 2 so that if the message to broadcast is m. By validity processes in group 2 should deliver m'.
Scenario C: All processes in group 1 are malicious and act to processes in group 2 so that if the message to broadcast is m', and act to processes in group 3 so that if the message to broadcast is m. Processes in group 3 cannot distinguish this scenario from scenario A, and would deliver m. Processes in group 2 cannot distinguish this scenario from scenario B, and would deliver m'. This contradicts the weak agreement property.

(b) Here is the algorithm:

- Each process has a local variable m. Initially $m =$SF.
- Sender in beginning of round 1: Sends message to broadcast to everyone, signed by itself, as (M, P_{sender}).
- Any crusader in round 1 (including the sender after broadcasting the message): If received message (m', P_{sender}) from sender, sets $m = m'$.
- Any crusader i in round 2:
  Send $((m, P_{sender}), P_i)$ to everyone
  Receive round 2 messages
  If received at least one valid message $((x, P_{sender}), P_.)$ such that $x \neq m$ and (x, P_{sender}) is not a valid message from sender (does not hold a valid signature, etc), then deliver SF
  If received at least one valid message $((m, P_{sender}), P_.$ deliver SF
  Otherwise deliver SF

The correctness:

Termination is obviously satisfied.

Validity: If sender correct, any correct process receives a valid message of that form (at least from itself) and delivers the sender's message.

Weak Agreement: If sender sends the same message to every correct process, processes cannot deliver different messages. If sender sends different messages to correct processes, then they see the valid message from each other with another message from sender with a valid signature from sender, and deliver SF. So weak agreement cannot be violated.

3. (a) Yes. Consider a run of the algorithm that we know satisfies agreement. Suppose by way of contradiction that it does not satisfy uniform agreement. It means that there is an execution which in there exists a crashed process p, that before crashing has decided something different from the correct processes. But now we can change the run such that this process p does not crash, but all its messages to others after the old crash time are delayed to after everyone has decided or crashed. So the new messages cannot affect other processes. We also need to change the decisions of failure detectors. All they need to do is that after the time everyone is crashed or decided, they change they idea about p and do not suspect it anymore. So we have reached to an execution that no process feels different, but p do not crash after deciding anymore. So by agreement, it should have decided the same as what correct processes decide, which is a contradiction to what we assumed.

   (b) No. We know that A on a $\diamond P$ does not violate agreement. Assume by way of contradiction that A using arbitrary failure detectore D may violate agreement. Now we construct a failure detector that behaves like D until every process has decided, after that it behaves like $\diamond P$. By termination property of consensus, our new failure detector eventually acts like $\diamond P$. Means that it is a $\diamond P$ failure detector itself. But now, as it behaves like D until everyone decides, by our assumption processes may violate agreement. But it is a $\diamond P$, and we know that processes should now violate agreement, which is a contradiction.

   (c) - (a) Yes. The proof is pretty like the case (a). Consider a run of the algorithm that we know satisfies agreement. Suppose by way of contradiction that it does not satisfy uniform agreement. It means that there is an execution which in there exists a crashed process p, that before crashing has decided something different from the correct processes. First we alter the decisions of failure detectors about p. All they need to do is that the postpone their decision that p is crashed, to the time after everyone is crashed or decided. This execution still should solve consensus, and the decision of p is not different. But now we can change the run such that this process p does not crash, but all its messages to others after the old crash time are delayed to after everyone has decided or crashed. So the new messages cannot affect other processes. Moreover p would now should in any failure detectors suspects list, but it cannot affect any process too, as we showed that FDs would add p to their lists just after everyone has decided or crashed. So we have reached to an execution that no process feels different, but p do not crash after deciding anymore. So by agreement, it should have decided the same as what correct processes decide, which is a contradiction to what we assumed.

- (b) Yes this time. From A, construct A' this way: In A', every process at every time behaves exactly like A, unless its failure detector has every one in its suspects list. In this case A' decides its own value.
  As in perfect FD no process is suspected before it crashes, then A' behaves exactly like A, because perfect FD never returns a suspects list full of all processes to a alive process. Then A' solves consensus using perfect FD too. But it's pretty easy to see how arbitrary FD can ruin A'. It just need to give every process a suspects list containing all processes at the very beginning. Then A' forces every process to decided it's own initial value, which would most of the times violate agreement.

4. (a) Yes.
   Any process i that is crashed at time τ, would not send any "I am alive" message after time τ. As there are at least $n - t$ correct processes, then every correct process j would eventually enter a new round of the failure detector algorithm (step i), and eventually finishes that round (completeing step iii), but does not recieve a "I am alive" from i, so would put i in it's suspects list. This would repeat forever and i would remain in every correct process's suspects list forever.

   (b) Yes.
   At each time τ any correct process i suspects exactly t processes, which all can be correct processes. There are at least $n - t$ correct processes, t of them can be suspected, then at least $n - 2t > t$ correct processes are not suspected, which is a majority.

   (c) No.
   If one could convert this to $\diamond S$, then it could be used to solve consensus. But we already know that consensus cannot be solved in this model, with $t \geq 1$. If $t = 0$, then you will not need any failure detector at all.

5. Yes.
   It is easy to use less-than-safe registers to build safe registers. All we need to do is to put in place of any safe register, n less-than-safe registers, one for each process to read. This way as no two process read the same register, there is no two concurrent read operations. Means that less-than-safe registers would behave like safe registers. And we already know that putting n safe registers in place of one cannot corrupt anything (for example in the way we used 1-reader registers to construnt m-reader registers).

6. (a) True.

Mutual Exclusion: Assume by way of contradiction that at some time two processes i and k happen to be both in CS. Consider the first such time and observe the following relations. Apparently i has read some value other than 2 for $control_k$ in line 8, so:

$$i.6[W(control_i, 2)] \longrightarrow i.8[R(control_k, \neq 2)] \dashrightarrow k.6[W(control_k, 2)]$$

By symmetry, the same is true for k:

$$k.6[W(control_k, 2)] \longrightarrow k.8[R(control_i, \neq 2)] \dashrightarrow i.6[W(control_i, 2)]$$

Mixing the two and by $\longrightarrow \dashrightarrow \longrightarrow$ we get:

$$i.6[W(control_i, 2)] \longrightarrow k.8[R(control_i, \neq 2)]$$

which contradicts with what we already know:

$$k.8[R(control_i, \neq 2)] \dashrightarrow i.6[W(control_i, 2)]$$

Note: The above argument works for safe registers too.

Deadlock Freedom: Assume by way of contradiction that a deadlock happens. Means, there is a set T of processes that loop forever in lines 3 to 8, and any other process is forever in NCS section. It is easy to see that for every process $i \in T$, $control_i \geq 1$ and for every process $i \notin T$, $control_i = 0$. Consider the process in T with smallest process id, call it k. k cannot reach the **goto** statement in line 5, so always proceeds to loop in line 8. But for k to reach the **goto** in line 8 infinitely many times, there should be processes with $control_j = 2$ infinitely many times; but it cannot be true, as any process j with $control_j = 2$ would see $control_k = 2$ in line 8 and goto L0, set $contorl_j = 1$ and forevery look in lines 3 to 5, as $k < j$ and $control_k \neq 0$. Then for every process in T other than k, there is a time after which it's *control* cannot be 2. Then k is the only process with control 2, and would eventually reach CS, contradicting that we assumed there is a deadlock already happened.

(b) It is easy to see how a deadlock can happen. As everytime some process is writing to its *control* register, others can read an arbitrary value. Here is an example for two processes:

- process i forever does these read and write operations:
  - in time slot $1 + 12k$ to $4 + 12k$ it is writing: $i.3[W(control_i, 1)]$
  - in time slot $5 + 12k$ to $8 + 12k$ it is writing: $i.6[W(control_i, 2)]$
  - in time slot $9 + 12k$ to $12 + 12k$ it is reading: $i.8[R(control_j, 2)]$
- process j, initially does $j.3[W(control_j), 1]$ in time slot 1 to 2. Then forever does these read and write operations:
  - in time slot $3 + 12k$ to $6 + 12k$ it is reading: $j.5[R(control_i, \neq 0)]$
  - in time slot $7 + 12k$ to $14 + 12k$ it is writing: $j.3[W(control_j, 1)]$

It can be easily checked that the above read and values are valid ones in a safe register model. And it is a deadlock.