Concurrent Computing December 15, 2025

Non-graded Midterm (2024's final) Solution

Disclaimer: most solutions presented here are more detailed that what is expected by students. This is especially true for the longer ones like 1.2, 1.3, 3.2 and 4.3 where we don't even expect 50% of that. For example, for proofs, we expect you to give us just enough details to allow us to be sure that you understand the path that the full proof should take. So you can skip some trivial steps here and there, but if you skip non-trivial steps or too many steps at once, or if what you write is inaccurate, we will likely penalize.

Question 1

1. When a read is concurrent to at least one write, a regular register makes no guarantee on the value on the value being returned by the read, while a regular register guarantees that the read returns either the "old" value (the one written by the last completed write before the read), or one of the values that are written concurrently.

2. No, `R` would not be atomic. Since `REG` is unsafe, a `REG.read` (line 14) that is concurrent to a `REG.write` (line 9) can return any (invalid) timestamp and any (invalid) value. If this invalid timestamp is higher than `last_v`, a `R.read` will return this invalid value, violating atomicity.

   Here is an example execution. Assume there was no prior operations, and `R.write(7)` is run concurrently to `R.read()`. More specifically, the `REG.write((1, 7))` (from `R.write(7)`) is concurrent to `REG.read()` (from `R.read()`). Since the read of `REG` is concurrent to a write, anything can be returned. Let's assume `REG.read()` returns `(42, 99)`. This means the check on line 14 will pass, and `R.read()` will return 99. Since 99 is neither the previous value ($\perp$) nor a value being written concurrently (7), `R` can neither be atomic nor regular.

   "Bonus" note: If the invalid timestamp read at line 14 is lower than `last_v`, a stale value might be returned (e.g. if multiple writes happened since the last read) which would also violate atomicity.

3. Yes, `R` would be atomic. Since `REG` is regular, in line 14, `REG_sn` is always set to the timestamp of the last completed `R.write` or one of a concurrent `R.write` (if any). The values associated with these timestamps are all fine to return, unless a previous read already returned a newer one, in which case we can simply return the newer one (since it's write must be concurrent to this new read too, as we would otherwise not have read an older timestamp). This what the the if condition ensures. (Note that this transformation only works for SRSW registers.)

   (Note: This is not a rigorous proof, but should be more than enough since the question only asks to justify)

Question 2

1. Serializability ensures committed transactions happen atomically, that is, as if they happened at a single indivisible point in time within their execution. In other words, there exist a serial order of the committed transactions that matches the real-world order (if T1 happened before T2, it must be before T2 in the serial order) and would produce the same result.

2. Opacity simply extends Serializability to aborted transactions. That is, Opacity ensures Serializability but also that even aborted transactions observe a consistent state of memory.

3. If Opacity is ensured, this means the `Total` is always 1000, and nothing wrong can occur.

4. If only Serializability is ensured, this mean a transaction (one that would abort) could compute a `Total` of 0 (or any value between 0 and $1000 * M$), which can lead to an error due to a division by 0.

Question 3

1. Agreement is violated. Here is an example execution: Assume 1st process runs `propose(v)` alone, and thus return `(commit, v)`. Then, if a 2nd process comes after and runs `propose(u)` with $u \neq v$, this second process will return `(adopt, u)`, violating Agreement.

2. A simple fix is to add the following two lines between line 16 and 17:

   1 **if** some non-$\perp$ value in B is $[true, u]$ for some u **then**
   2 **return** $(adopt, u)$

   Now let's prove every property:

   - **Validity:** the values returned are either proposed by the current process, or a non-$\perp$ value from B. Since values written in B are all proposed values, Validity follows.
   - **Commitment:** If every process that proposes does so with the same value v, then every non-$\perp$ values in A are this same value and every process takes the first branch on line 10. This implies that all non-$\perp$ values in B are $[true, *]$, thus every process takes the first branch on line 16 and returns `(commit, v)`.
   - **Agreement:** Let's assume by contradiction that a process p outputs `(commit, v)` while a process q outputs `(*, u)` with $u \neq v$.

     We first note that each index in the snapshot objects A and B are only ever written at most once (values are never overwritten nor erased). We also note that processes read A or B only after having written to it, thus they must read at least one non-$\perp$ value.

     Since p committed v, this means that at some point, non-$\perp$ values in B were all $[true, *]$, and thus the first value ever written to B was of the form $[true, *]$. This also means that p must have wrote $[true, v]$ at line 10, and thus that non-$\perp$ values in A were all v at some point, so the first value written in A was v.

     Now let's see where q could return:

     - If q returns from line 16, it means all non-$\perp$ values in B where all $[true, *]$, thus it can not have run line 12 and thus must have seen only u in A, which is impossible since v was the first value written in A, a contradiction.
     - If q returns from the new branch (defined above), it must have seen some $[true, u]$ in B, thus a process r that proposes u must have run line 10, and must have seen only u in A, which is impossible since v was the first value written in A, a contradiction.
     - If q returns from line 20 (previously line 18 in the original algorithm), it means q observed no $[true, *]$ in B, which is impossible since a value of the form $[true, *]$ was the first value written in B, a contradiction.

     Since all cases lead to a contradiction, Agreement must be true.
   - **Termination (Wait-freedom):** There are no loops or recursion, and all operations called are wait-free thus this implementation is wait-free.

Question 4

1.
   - In obstruction-freedom, a process that takes steps is only guaranteed to make progress if it runs alone for long enough.
   - In lock-freedom, it is guaranteed that at least one process that takes steps makes progress, no matter what (even this process does not run alone).
   - In wait-freedom, no just one, but all processes that take steps make progress, no matter what.

2.
   - **Agreement** guarantees that no two processes decide differently.
   - **Validity** guarantees that no process decide a value that was not proposed.

3.
   - **Validity:** Since we return *est*, we can simply prove that *est* always contains a proposed value. We observe that *est* is first set to v, a valid value. Then, in every iteration, assuming *est* was a proposed value at the end of the previous iteration, we only write proposed values in the snapshot object $S[r]$, thus *est* is still a proposed value in line 12. In line 13, we thus only use consensus-proposed values to the commit-adopt propose function, and by the Validity property of the commit-adopt object, it must thus return a concensus-proposed value. Thus, since *est* is always a proposed value, we always return a proposed value.
   - **Agreement:** If a process decides v in round r, this means $C[r]$ returned $(commit, v)$ to that process, and thus can only have returned $(commit, v)$ or $(adopt, v)$ to the other process by the Agreement property of commit-adopt. If it received $(commit, v)$ it will decide v immediately. Otherwise, this means that in round $r+1$, the other process will write v in $S[r+1]$ and then propose v to $C[r+1]$ and since only v was proposed to $C[r+1]$ it must thus obtain $(commit, v)$ by the Commitment property of commit-adopt, so it will decide v.
   - **Obstruction-freedom:** If a process runs alone for long enough, it will reach a round that the other process didn't reach, because a single round has no loops inside, and all objects involved are wait-free. Then, once this process runs a round alone, by the Commitment property of the commit-adopt object it must obtain $(commit, v)$ and thus decide v.

4. FLP states that wait-free consensus can not be implemented using only registers, but all the objects used in this implementation can be implemented using only registers, thus it can not implement wait-free consensus.

   Alternative answer: We can construct an execution where the processes never return: process 1 proposes 1, and process 2 proposes 2. In every round r, process 1 executes line 9 and 10 before process 2 executes line 9, thus process 1 sets *est* to 1 in line 12, while process 2 sets it to 2. They both run $C[r].propose(est)$ concurrently (line 13), and obtain $(adopt, 1)$ and $(adopt, 2)$ respectively. They thus can not commit, and proceed to the next round with the same *est* values as before, looping indefinitely.

5. In line 12, instead of always taking the max, we can flip a coin and take the min if $coin.flip()$ return 1, and the max if it returns 0. The second process that reaches line 10 will necessarily have the two values in the snapshot v, and thus will decide randomly which value of the two values it proposes to the commit-adopt object. There is thus at least a 50% chance that both processes propose the same to the commit-adopt object, and by its Commitment property, thus guarantees that they will get $(commit, est)$ at least 50% of the time, thus having at least 50% chance to decide in every round.

Question 5

1. If a correct process invokes leader, then the invocation returns and eventually, some correct process is permanently the only leader.
2. We can reuse commit-adopt in a similar fashion as Question 4, as follows:

```
1  Shared state:
2  An infinite array of commit-adopt objects, C[∞].
3  A register decided, initialized to ⊥.
4
5  procedure propose_i(v)
6      est ← v
7      r ← 0
8      while decided.read() = ⊥ :
           /* Wait to be leader (or to read something from decided)        */
9          if not Leader◇.leader() then
10             continue
11
12         (dec, est) ← C[r].propose(est)
13         if dec = commit then
               /* help non-leaders before deciding                         */
14             decided.write(est)
15             return est
16
17         r ← r + 1
       /* Return if anything was decided                                   */
18     return decided.read()
```

There are many possible alternatives. See the slides on "Consensus with Timing Assumptions" for a detailed solution that uses timestamps instead of rounds of commit-adopt.

Question 6

1. This swap object has two states, 1 and 2 (01 and 10 in binary), which basically depends on the amount of *swap* calls, modulo 2. It can thus be easily implemented with a *Counter* object, which, as seen in class, can be implemented using registers. Since registers only have consensus number 1, so does this SWAP object.

2. To show that this object has consensus number 2, we can implement consensus between two processes using this object and any amount of registers. Here is an example algorithm:

```
initially: S = SWAP(R = 1), Array a = [⊥, ⊥]
/* The identifier of the process is i (starting from 0)                    */
1  procedure propose_i(v)
2      a[i] ← v
3      if i = 0 then
4          S.decrement()
5      else
6          S.swap()
7
8      if S.read() = 0 then
9          return a[0]
10     else
11         return a[1]
```

   This algorithm works as follows: R is initialized to 1. Thus, if process 0 runs $S.decrement()$ before process 1 runs $S.swap()$, R is decremented to 0 and remains 0 forever after (the $S.swap$ does nothing on 0), thus they can only decide $a[0]$. Otherwise, if process 1 runs $S.swap()$ first, then R is changed to 2 (10 in binary), thus R will remain non-0 even if process 0 calls $S.decrement()$ once, and they can only decide $a[1]$.