SLR210: Solutions for Quiz 1

1 ABD algorithm

1.1 Atomicity violations

Consider a run of the algorithm in the slides in which the writer starts a *write* operation by sending (v, t), a new value v with an incremented timestamp t, to every other process. Suppose that the message is first received by a sible reader p_i (the *write* operation is still incomplete).

Now let p_i execute a *read* operation and return v (as this is the most recent value it sees. After the *read* operation completes, let another reader p_j execute a *read* and suppose that this operation reaches a quorum of processes that does not include the writer or p_i (this is possible if we have at least 5 processes).

Therefore, the second read will have to return an older value, which results in a *new-old inversion* violating linearizability.

1.2 Multiple readers

A standard solution to accommodate multiple readers is to let the readers communicate with each other (the reader must write). Before returning the read value v, the reader should "write" the orresponding tuple (v, t) back to the system. This way, a subsequent *read* operation will return v or a more recent value.

Essentially, the reader executes the code of the writer: it sends (v, t) to everybody and waits until a quorum of processes acknowledges that they received the message. Only after this it is safe to return from the *read* operation.

1.3 Multiple writers

When multiple processes are allowed to write to the implemented register, they cannot use independently maintained local timestamps. Indeed, a slow process executing its first *write* operation may have a timestamp that is lower than the timestamp used for a complete operation The readers will not be able to distinguish the new value from an old one.

To resolve this issue, we can adopt the apporach of Lamport's *doorway protocol* (part of his Bakery mutual-exclusion algorithm). A writer first *reads*, i.e., collects the values stored at a quorum of processes, computes the highest sequence number t used so far, and adopts $t + 1$ as its new sequence number.

To break the symmetry among values written with the same sequence number, we attach the writter's identifier to the timestamp. Assuming that timestamps (t, i) and (t', j), where t and t' are sequence numbers and i and j are writers' identifiers, are compared *lexicographically*, all written values are given *distinct* and *totally ordered* timestamps. Moreover the total order respects the real-time order of *write* operations: the timestamp used by a write operation W is lower than the timestamp used by any write operation that starts after W completes.

The total order on the timestamps used by the writers in a given history can be used to construct a *linearization*: the *write* operations are put in the order of their timestamps and the

complete *read* operations are put after the corresponding writes, respecting the real-time order among them.

2 Read-optimized quorum systems

Let P be the set of processes and consider (W_P, R_P):

- $W_p = P$
- $R_p = \{\{p\} | p \in P\}$.

Obviously (W_P, R_P) respects the quorum safety property (any element of W_p intersects with any element of R_P). Assuming that no failures occur, (W_P, R_P) is also live.

To write a value, the writer must reach all the processes, but reading only requires reaching one.

> It is often argued that in usual workloads of storage systems, read operations are invoked much more often than write operations. Read-optimized quorums can be very useful in reliable geographically distributed systems. A write can be very slow, as it has to hear from every replica. But the read will terminate as soon as the closest replica responds. Of course, if a single process fails, the write will never terminate.

3 Lattice Agreement

3.1 One-Shot Lattice Agreement

Recall that in a regular specification of an atomic snapshot, a process p_i can only update its own position i in the snapshot memory. Therefore, any two updates, concurrently applied by p_i and p_j *commute*: they result in the same state, regardless of the order in which they are applied. In the one-shot case, a process performs a single update followed by a snapshot. We assume that, intitially, every position of the snapshot memory contains a special value $\perp$. Let V be the set of values that can be used by update operations as arguments, and suppose that $\perp \notin V$.

It is easy to solve a single instance of lattice areement using a (one-shot) atomic snapshot: every process uses its input in the lattice as the argiment of its update operation, then takes a snapshot, and outputs a join on the non-$\perp$ elements in the returned vector. As the sets of non-$\perp$ elements snapshot taken by different elements are related by containment, the resulting lattice elements are realted by $\sqsubseteq$.

For the other direction, let us define the lattice on sets of the type $\{(v_1, i_i), \ldots, (v_k, i_k)\}$, where $v_j \in V$ and $i_j \in \{1, \ldots, n\}$. The partial order is then simply the set inclusion and the join operator—the set union.

To execute *update*(v_i) followed by *snapshot*(), p_i proposes $\{(v_i, i)\}$ to lattice agreement and returns the vector of values, where each position i contains v_i if (v_i, i) is incuded in the output of lattice agreement, and $\perp$ otherwise. Note that, as only the single update performed by p_i can propose a value $(-, i)$, there can be at most one value (v, i) in any output of lattice agreement.

Convince yourself that the solution indeed implements an atomic snapshot.

3.2 Long-Lived Lattice Agreement

Again, generalized lattice agreement can be trivially implemented using (long-lived) atomic snapshots. Whenever the t-th value v is *received* by a process p_i, it performs *update*(v, t). To *learn* a new value, p_i takes a snapshot and simply returns the join of the most recent values in the returned vector.

For the other direction, we define a lattice on the sets of the type $\{(v_1, t_1, i_1)), \ldots, (v_k, t_k, i_k)\})$, where each tuple (v_j, t_j, i_j) corresponds to the value v_j written to position j with sequence number t_{i_j}. (v_i is the argument of the t_i-th update performed by p_i). The origin of the lattice is defined as $(0, 0, 1), \ldots, (0, 0, n)$: every position $j \in \{1, \ldots, n\}$ stores the initial value 0 written with the intitial sequence number 0.

Again, the partial order $\sqsubseteq$ and the union operator $\sqcup$ are defined as theset inclusion and the set union, respectively.

To execute *update*(v_i), p_i increments its locally maintained sequence number s_I and *receives* $\{(v_i, s_i, i)\}$ for the generalized lattice agreement and waits until it learns a value that contains (v_i, s_i, i). To execute *update*(v_i), p_i *receives* $\{(\bot, s_i, i)\}$ for the generalized lattice agreement, waits until it learns a value that contains $(\bot, s_i, i)$ and returns the vector where each position j contains the value v_j in the tuple (v_j, t_j, j) with the highest sequence number t_j for j. Again, as only p_j is allowed receive elements of the kind $\{-, -, j)\}$ for the generalized lattice agreement, and each new received value for j contains a distinct sequence number, the vector above is well-defined.

Convince yourself that the solution indeed implements a long-lived atomic snapshot.

> Note that the two atomic-snapshot implementatons described above assume that a given position can only be modified by a dedicated process. Is it possible to extend the algorithms to get a *generalized* atomic-snapshot memory which maintains a shared vector of m positions and allows any process p_i, $i \in \{1, \ldots, n\}$, to perform *update*(v, j) on any position $j \in \{1, \ldots, m\}$?