Problem 1. As given in the lecture, a queue is a shared object with two operations: enq(x) and deq(). Now we augment the queue object with an operation peek that returns but does not remove the first element in the queue. You task is to find an algorithm that implements consensus among \( n \) processes using the augmented queue object and atomic registers for an arbitrary \( n \). (You may use any number of the augmented queue objects and atomic registers.)

Solution

Here is a simple consensus protocol. As usual, threads share a public announce[] array. In the propose() method, the thread with index A writes its proposed value to announce[A], then calls enq(A), and then calls peek(), which returns B. Finally, thread A decides the value in announce[B]. This is correct because if B is the first index to be enqueued, later enq() calls will not change Bs position, and B stored its value in announce[B] before calling enq(B).

Problem 2. A queue is implemented using two objects: fetch&inc (augmented with operation read) and swap as follows. A swap object is a shared object with an operation swap(v) that atomically replaces the current value of the object with \( v \) and returns the prior value. (The swap object has also operation read.)

Using a fetch&inc object head, initialized to 0.

Using an array of swap objects (of infinite length) items, initialized to ⊥.

enq(x):
   slot = head.fetch&inc();
   items[slot].swap(x);
   return ok

deq():
   while(true) do
      limit = head.read();
      for j = 0 to limit - 1 do
         y = items[j].swap(⊥);
         if (y is not ⊥) then
            return y
You may assume this queue is linearizable, and wait-free as long as deq() is never applied to an empty queue.

Consider the following sequence of statements:

- Both fetch&inc and swap objects have consensus number 2 (i.e., there is an algorithm that implements consensus using the shared object and atomic registers among two processes, while there is no algorithm that implements consensus using the shared object and atomic registers among three or more processes).

- We can add an operation peek() simply by taking a snapshot of the queue (using the snapshot algorithm studied in previous lectures) and returning the first item in the snapshot.
• Using the protocol devised for Problem 1, we can use the resulting queue to implement consensus among \(n\) processes for an arbitrary \(n\).

We have just implemented consensus using only objects with consensus number 2. Your task is to identify the faulty step in this chain of reasoning, and explain what went wrong.

Solution

The problem is that the snapshot does not tell you the queue’s actual state. The operation of making room for the head and then inserting a value into it is not atomic. Start with an empty queue. The first thread does an enqueue and increments head to make room for its item, but does not write its value, leaving that spot null. A second thread then runs and completely finishes enqueuing its value. It then takes a snapshot and finds the space that the first thread had made, which is still null. The second thread has no choice but to decide on its own value — the only value in the queue (waiting for the first thread would break the termination property if the first thread has crashed). Then the first thread wakes up, finishes writing its value and takes a snapshot to see its value is first and decides on it, a different value. So consensus is not reached.

Problem 3. A \(k\)-set-agreement object is a generalization of a consensus object in which processes could decide up to \(k\) different values. Formally, \(k\)-set-agreement is defined as follows. It has an operation \(\text{propose}(v)\) that returns (or we say decides) a value, which satisfies the following properties:

1. Validity: Decided values are proposed values.
2. Agreement: At most \(k\) different values could be decided.
3. Termination: Every correct process eventually decides a value.

A \(k\)-simultaneous-consensus object is another generalization of a consensus object in which processes could decide \(k\) values simultaneously. Formally, \(k\)-simultaneous consensus is defined as follows. It has an operation \(\text{propose}(v_1, \ldots, v_k)\) that returns (or we say decides) a pair \((\text{index}, \text{value})\) with \(\text{index} \in \{1, \ldots, k\}\), which satisfies the following properties:

1. Validity: If a process decides \((i, v)\), then some process proposed \((v_1, \ldots, v_k)\) with \(v_i = v\).
2. Agreement: If two processes decide \((i, v)\) and \((i', v')\) with \(i = i'\), then \(v = v'\).
3. Termination: Every correct process eventually decides a value.

Your task is to show that \(k\)-set-agreement and \(k\)-simultaneous-consensus are equivalent. That is, you have to show that one implements the other.

Hint: When implementing \(k\)-consensus using \(k\)-set-agreement, an algorithm that solves the problem is the following:

1: \textbf{function} \textsc{kSC.PROPOSE}(v_1, \ldots, v_k)
2: \hspace{1em} V_i \leftarrow [v_1, \ldots, v_k]
3: \hspace{1em} dV_i \leftarrow \textsc{kSA.PROPOSE}(V_i)
4: \hspace{1em} REG[i] \leftarrow dV_i
5: \hspace{1em} \text{snap}_i \leftarrow \text{REG.snapshot}()
6: c_i \leftarrow \text{number of distinct (non-\(\perp\)) vectors in \text{snap}_i}
7: d_i \leftarrow \text{minimum (non-\(\perp\)) vector in \text{snap}_i}
return \langle c_i, d_i \rangle \]
end function

Where \( \text{REG}[0, \ldots, n-1] \) is an array of single-writer multi-readers atomic registers initialized at \( \perp \). Processes write atomically a vector of values in their register (Line 4). \( \text{REG.snapshot()} \) returns an atomic snapshot of this array of registers. Consequently, \( \text{snap}_i[0, \ldots, n-1] \) is an array of vectors, possibly containing \( \perp \) values for some indices. We suppose that there is an order on the set of values that can be proposed, and we use the induced lexicographic order on vectors at Line 7.

Your task is then to (1) prove that the algorithm above implements a \( k \)-simultaneous consensus from \( k \)-set agreement objects and atomic registers; and (2) find an algorithm that implements a \( k \)-set agreement object using \( k \)-simultaneous consensus objects and atomic registers.

**Solution**

We will show that the \( k \)-set agreement problem and the \( k \)-consensus problem are equivalent. To do that we will show two wait-free constructions, one in each direction. Both constructions are independent of the number of processes.

**From \( k \)-consensus to \( k \)-set agreement** A pretty simple wait-free algorithm that builds a \( k \)-set agreement object (denoted KSA) on top of a \( k \)-consensus object (denoted KC) is described below. The invoking process \( pi \) calls the underlying object KC with its input to the \( k \)-set agreement as input, and obtains a pair \( \{c_i, d_i\} \). It then returns \( d_i \) as the decision value for its invocation of KSA.set.propose.k.

```c
KSA.set_propose_k(vi)
{
    \{c_i, d_i\} = KC.sc_propose_k(vi);
    return d_i;
}
```

**Proof.** The proof is straightforward. The termination and validity of the \( k \)-set agreement object follow directly from the code and the same properties of the underlying \( k \)-consensus object. The agreement property follows from the fact that at most \( k \) values can be decided from the \( k \) consensus instances of the \( k \)-consensus object.

**From \( k \)-set agreement to \( k \)-consensus** We first need to transform the vector version of \( k \)-consensus to a scalar version. It is easy to see that the vector version and the scalar version are equivalent (i.e., each one can implement the other one) when the size of the set \( I \) of input values is the same in both problems.

1. **From the vector version to the scalar version:** Implementing the scalar version from the vector version is trivial. Let \( vi \) be the value proposed by \( pi \) in the scalar version. The value it proposes to the vector version is simply the vector \([vi, \ldots, vi]\).

2. **From the scalar version to the vector version:** The algorithm described below implements the vector version from the scalar version. A process \( pi \) first proposes the vector input \( i \) (that contains its vector proposal) to each consensus instance of the underlying scalar version of the \( k \)-consensus problem. It then obtains a pair \( \{c_i, wi\} \) and decides on the pair \( \{c_i, d_i\} \) where \( d_i \) is the value in \( wi[c_i] \) (i.e., a value proposed by a process to the \( c_i \)-th consensus instance). The proof is easy and left to the reader. It is also easy to see that the size \( n \) of the set \( I \) of input values is the same in the vector version and the underlying scalar version.
KSC.sc_propose_k(vi1, ..., vik) //vector version
{
    inputi = [vi1, ..., vik];
    {ci, wi} = KSC.sc_propose_k(inputi); //scalar version
    let di = wi[ci];
    return ci, di;
}

A wait-free algorithm that constructs a $k$-consensus object KC from a $k$-set agreement object KSA is described below. (|snapi| denotes the number of elements in snapi.) In the algorithm, the processes first go through a $k$-set agreement object to reduce the number of distinct values to at most $k$ (line 01). Then, each process $p_i$ (1) posts the value it has just obtained in the cell Reg[i] of the shared memory (initialized to null), and (2) takes a snapshot of the whole shared memory (line 03). Finally, a process $p_i$ returns the pair $\{ci, di\}$ where the consensus instance $ci$ is defined as the number of values in the set returned to $p_i$ by its snapshot invocation, and $di$ is the minimum value in that set.

KC.sc_propose_k(vi)
{
    dvi = KSA.set_propose_k(vi);
    Reg[i] = dvi;
    snapi = Reg.snapshot();
    let ci = |snapi|; let di = min value in snapi;
    return ci, di;
}

Proof. The code is wait-free since there are no loops and both the $k$-set agreement and the snapshot operations are wait-free. The validity follows from the fact that all the values in the algorithm originate from process inputs. Since the snapshots by the different processes define a linearizable sequence ordered by containment, they also define a non-decreasing sequence when we consider the size of the snapshots returned to the processes. Therefore, there is a unique snapshot value of a given size and hence the minimum value in each snapshot of a given size is unique. Thus there are at most $k$ distinct snapshot sizes, each with its unique minimum value. Hence, there are at most $k$ distinct outputs returned and any two processes that return a pair with the same snapshot size (same first coordinate) have the same value associated with it, which proves the agreement property of the $k$-consensus.

Note: The algorithm presented in the problem statement, is the combination of the two constructions.