CSC2221: Introduction to Distributed Computing **Fall 2017**

Lecture 7: Implementation of Primitives

Lecturer: Faith Ellen *Scribe: Lily Li*

7.1 Multi-reader Registers (AW 10.1)

Theorem 7.1 *(10.3 Multi-reader Single-writer Register from Single-reader Single-writer Registers.) In any wait-free simulation of a single-writer multi-reader register from any number of single-writer single-reader registers, at least one reader must write. (As shown below, the restriction is actually stronger.)*

Proof: We proceed by contradiction and suppose that there exists a simulation for register R where no process writes. Let p_w be the writer and p_1, p_2 be two readers. Assume that p_w wants to write a 1 into R. p_w performs some sequence low-level writes $w_1, ..., w_k$ to a set of registers R_1 read only by R_1 and a set of registers R_2 read only by R_2. Consider what happens if there was a read by p_1 and a read by p_2 between low-level writes. Initially these reads must return 0 — p_w have not started reading yet. At the end both reads must return 1 — p_w finished writing. Somewhere in the middle there exists two indices j_1 and j_2 such that the read by p_1 between w_{j_1} returns 1 (while all previous reads have read 0) and similarly for p_2. Observe that $j_1 \neq j_2$ since w_{j_1} and w_{j_2} are writes to registers in R_1 and R_2 respectively and are distinct. WLOG assume that $j_1 < j_2$ then schedule a read by p_1 before a read by p_2 within the interval w_{j_1}, w_{j_1+1}. Then p_1 reads a 1 and then p_2 reads a 0 but the read by p_1 occurred before the read by p_2. ■

The solution? Sequence numbers (or equivalently, timestamps).

7.2 Multi-reader Register (AW 10.2.2)

Implementation of a multi-reader single-writer register from single-reader single-writer registers (AW 10.2.2). One naive way would be to use a collection of single-reader registers one per reader so that $val[i]$ can be read by p_i. See Algorithm 1. Unfortunately this implementation is not linearizable (can you see why? Hint: write 1 between two complete reads which have different values).

Algorithm 1 Implementation of Multi-reader Single-writer

```
1: WRITE(R, v)
2: for all readers p_i do
3:     write(val[i], v)
4: end for
5:
6: READ(R) by p_i
7: return val[i]
```

Theorem 7.2 *In any implementation of a MRSW register from SRSW registers, all but one process must write.*

Proof: Consider solo execution of $WRITE(R, 1)$ by p_w starting from initial configuration $R = 0$. Consider the view from another processor p_i: $C_0 s_1 C_1 \cdots C_{j_i} \cdots s_k C_k$. Obviously reads from C_0 return 0 and reading from C_k returns a 1. Note we cannot have the situation where read from C_i return 1 and read from C_{i+1} returns 0 (why?). So there exists some step s_{j_i} such that C_{j_i-1} reads 0 and C_{j_i+1} reads 1.

Suppose for two processors $p_i \neq p'_i$ that read without writing... (see above). ■

From the above algorithm we know that every reader except one must *write*. Here is algorithm we are going to develop: when p_w writes, it writes the value along with a sequence number $val[i]$ (initially, $(0, 0)$ for all i).

Algorithm 2 Implementation of Multi-reader Single-writer V2

```
1:  WRITE(R, v)
2:  seq ← seq + 1
3:  for i from 1 to n do
4:      write(val[i], (v, seq))
5:  end for
6:
7:  READ(R)
8:  (v, s) ← read(val[i])
9:  for for all i' ≠ i do
10:     (v', s') ← read(report[i', i])
11:     if s' > s then
12:         s ← s'
13:         v ← s'
14:     end if
15: end for
16: for all i' ≠ i do do
17:     write(report[i, i'], (v, s))
18: end for
```

For each pair (i, i') where $i \neq i'$ there is a SRSW register $report[i, i']$ (first index writer, second is reader) in which p_i report the largest sequence number it has seen together with its associated value (initially $(0, 0)$).

Next we need to show that this algorithm is Linearizable. We need some lemmas

Lemma 7.3 *(10.4) If op_1 finished before op_2 begins then op_1 occurs before op_2 in out Linearization π.*

Proof: Using the observations below lets consider the four cases for op_1 and op_2 for some admissible execution α. First consider if both op_1 and op_2 are writes, then they are linearized by definition. Next consider a write w that follows a read r (with timestamp T) in α but was placed after w in a linearization π. It must be the case that the write w' which generated T occurred after w. This implies that r occurred before w' but this is a contradiction. Next consider if a write w precedes a read r but was placed after r. Since the times are strictly increasing r must be placed after w. Finally consider two read operations r and r'. If the two jobs have different timestamps, the one with the smaller timestamp must have been generated first. If the timestamps are the same then we order interms of increasing starting times.

Yeah, please reread this proof. But look at the big idea: we are going to construct a linearization of the operations. Namely we will look at the sequence numbers! Writes obviously come one after the other and reads will be grouped by the sequence number. You should have some consistent (but not necessarily unique) ordering for all read with the same sequence number. ■

Further make the following observation (1) all writes have strictly increasing sequence number and (2) all reads have monotonically increasing sequence number.

Theorem 7.4 *(10.5) There is a wait-free implementation of MRSW register from $O(n^2)$ SRSW registers in which each read and write are performed in $O(n)$ steps.*

7.3 Multi-reader Multi-writer Register (AW 10.2.3)

From the our construction from previously we now have access to multi-reader single-writer registers. Let us see what needs to be done in order to turn this into a multi-reader multi-writer registers. The key idea is to use atomic-snapshots (atomic snapshots are great, they get around the pesky order business of using *collects*).

Algorithm 3 Implementation of Multi-reader Multi-writer

1: $WRITE(R, v)$
2: $v \leftarrow scan(s_1, ..., s_n)$
3: $t \leftarrow \max v_i, t_s : 0 \leq i \leq n-1$
4: $update(s_i, (x, t+1))$
5:
6: $READ(R)$
7: $v \leftarrow scan(s_1, ..., s_n)$
8: return v

Suppose you were a bit more ambitious and you wanted to use collect and write instead of just using a atomic-snapshot. Can you linearize the previous Algorithm 3, if you had *collect* instead of *scan* and *write* instead of *update*? Yes (of course the answer is yes), you would do it in the order of timestamps for $WRITE$s and $READ$s with timestamps and the order they finish.

Here is the formal argument: suppose op_1 finishes before op_2 begins. If op_2 is a $WRITE$, it must have larger timestamp than op_1 so is linearized after op_1. If op_2 is a $READ$ then two cases: (1) op_1 is a $WRITE$ then op_2 lin. after op_1 since op_2 can only read a $\geq$ timestamp than op_1 generated (2) op_1 is a $READ$, op_2 can only read a timestamp which is $\geq$ the timestamp read by op_1 thus it must be linearized after op_1.

How about a lower bound proof next?

Theorem 7.5 *Any such implementation has $\Omega(n)$ step complexity.*

Proof: In particular we are going to find a problem P which can be solved in $O(1)$ steps using MRMW registers, and prove it takes $\Omega(n)$ steps using MRSW registers. We will use *Approximate Agreement* (see below). ■

Example 7.6 *Each process p_i has a private input x_i if it doesn't crash, it must output value y_i. Further all processes are assumed to know an accuracy parameter ϵ. The solution must satisfy*

ϵ-Agreement: all output values are within ϵ of each other.

Validity: all output values must be between the largest and smallest input values.

We claim that Algorithm 4 is an algorithm for Approximate Agreement. Why does validity hold? (pretty straight forward: note $m_i = \min\{v_j : v_j \neq \perp\} \geq \{x_0, ..., x_{n-1}\}$ *and* $M_i = \max\{v_j : v_j \neq \perp\} \leq \{x_0, ..., x_{n-1}\}$ *so* y_i *works for all i) Why does* ϵ*-Agreement hold? (more complicated, but really if you think about it, it makes sense).*

Algorithm 4 Implementation of Approximate Agreement

1: $m \leq x_0, ..., x_{n-1} \leq M$
2: $\epsilon = \frac{M-m}{2}$
3: $X[0, ..., n-1]$ each entry initially $\perp$ # p_i can write to $X[i]$
4:
5: $X[i] \leftarrow write(x_i)$
6: $V \leftarrow collect(X)$
7: $m_i \leftarrow \min\{v_j : v_j \neq \perp\}$
8: $M_i \leftarrow \max\{v_j : v_j \neq \perp\}$
9: $y_i \leftarrow \frac{m_i+M_i}{2}$
10: return y_i

Notice here that out Algorithm 4 is for a particular value of ϵ*. What if we wanted* ϵ *to be smaller? (Supposing we are beginning with* $M = 1$ *and* $m = 0$*.) As always, with this sort of thing, iterating the procedure will allow you to halve the interval (you would just need a couple of collect objects, one for each iteration).*

Theorem 7.7 *For all* $\epsilon < 1$*, in every obstruction-free algorithm for approximate agreement using only MRSW registers, every process takes* $\geq n-1$ *steps in a solo execution starting from the initial configuration* C_0 *in which the input values of all processes are* 0 *(*$m = 0$*,* $M = 1$*).*

Thus worse-case step complexity of any wait-free algorithm is $\Omega(n)$*.*

Proof: Suppose for a contradiction that p_i takes $< n-1$ steps in its solo-execution from C_0. By validity, p_i must output 0. Let α be the history of its execution. There must be some process p_j, $i \neq j$, in which p_i did not read p_j's SWMR register.

Let C_1 be the configuration in which the input to all processes is 1. By validity, everyone, in-particular p_j, needs to output a 1. Let β be the history here.

Consider any configuration C in which p_i has input 0 and p_j has input 1. Observe that $C_0 \overset{p_i}{\sim} C$ and $C_2 \overset{p_j}{\sim} C$. Can you finish the reset of the argument?

Let $x_0, ..., x_{n-1} \in \{0, 1\}$ and $\epsilon = \frac{1}{2}$. Use two MWMR registers $W[0]$ and $W[1]$ both initially set to 0. The algorithm for each process p_i is in Algorithm 5.

Algorithm 5 Implementation of Approximate Agreement with MWMR Registers: code for process p_i

1: $WRITE(W[x_i], 1)$
2: **if** $READ(W[1-x_i]) = 1$ **then**
3: $\quad y_i \leftarrow \frac{1}{2}$
4: **else**
5: $\quad y_i \leftarrow x_i$
6: **end if**
7: return y_i

Again let us show that this algorithm works. First consider validity: if all processes have input v, then everyone outputs v. Please go through the agreement argument on your own. ■

7.4 Renaming

Our goal is to give each process a distinct name from $\{1, ..., m\}$ or $\{0, ..., m-1\}$ where $m \geq n$ the number of processes. There is only one operation: $GetName$. When a process performs $GetName$ it gets a name in $\{1, ..., m\}$ which is not held by any other process. The special case of **1-Shot Renaming** indicates that a process performs $GetName$ at most once. When a process performs $RelName$ its name is released and can be used by another process. In **Long-Lived Renaming** $GetName$ can only be performed by a process with no name. $RelName$ can only be performed by a process that has a name.

Algorithm 6 Implementation of Renaming

```
1: m = n
2: GetName() #Using Fetch and Increment (for 1-Shot Renaming)
3: return Fetch&Increment(F) #Does not support RelName()
4:
5: GetName() #Using Array of Test&Set Objects A[1, ..., n] where each entry is initially 0
6: for i from 1 to n do
7:     if Test&Set(A[i]) = 1) then
8:         return i
9:     end if
10: end for
11:
12: RelName() #By process with name i
13: reset(A[i])
14:
15: GetName() #Using Registers!
16: X is register holding original name; y ∈ {T, F} #Code for process with ID i
17: WRITE(X, i)
18: if y = T then go right then
19:     WRITE(Y, T)
20: end if
21: if X = i then
22:     capture the splitter
23: else
24:     go left
25: end if
```

Lets see why Algorithm 6 works. We show that it is impossible that in a reachable configuration a process p is in iteration i of $GetName$ but $A[i] = \cdots = A[n] = 1$.

We say that a process p is in **location** i if p has name i or is in iteration i of $GetName$ i.e. about to perform $Test\&Set(A[i])$. If p has no name then we say that p is in location 0. Consider the invariant: if a process is in location $i \geq 1$ then there is at least $i-1$ other processes in location $< i$ (proof of this invariant by induction; please go through the details).

(Doable if $m > 2n-1$, no if $m < 2n-2$ and it depends on number theoretic properties when $m = 2n-2$ — this was someone's PhD!) Lets do 1-Shot Renaming where $m = \frac{n(n_1)}{2}$. Uses m **splitters** (these can be constructed using two registers). Properties: if k process enter the splitter then at most one captures the splitter and at mot $k-1$ go left and at most $k-1$ go right (if one process goes into a splitter, it must be captured). Then here is the renaming algorithm. Big idea: Um... there is no pseudo-code, just a picture. Image a tree with splitters as nodes with height n and $i+1$ on level i.

7.5 Homework

Assume that $x_0, ..., x_{n-1} \in [0, 1]$ and $0 < \epsilon < 1$. Come up with an algorithm with $O\left(\log_2\left(\frac{1}{\epsilon}\right)\right)$ step complexity. How about an algorithm with $O\left(\log_2\left(\frac{\max\{|x_0|,...,|x_{n-1}|\}}{\epsilon}\right)\right)$ step complexity?