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

Lecture 8: Universal Constructions

Lecturer: Faith Ellen *Scribe: Lily Li*

8.1 Universal Constructions (AW 15.3)

An object O is universal if copies of O (and registers) can be used to implement any object (sequential operations).

Claim 8.1 CAS *is universal.*

Proof: Attempt 1: use one CAS object S which store the state of the object. Start with the initial state. Every time something happens, read S, perform the action locally, then try to write the result back to S. If the CAS was successful then return *true* otherwise try again. Note that this implementation is non-blocking but *not* wait-free (it could be better).

Attempt 2: Use an **announcement array** $A[0..n-1]$ of registers and each process p_i begins by announcing its operation and inputs in $A[i]$. S now has two fields: state and priority in $\{0, ..., n-1\}$. If the current state of S is (s, i) with the next op is given to p_i.

In the above p_i tries to perform its operation locally, then tries to CAS. If priority is j, then p_i reads $A[j]$ and sees if p_j has any operations to apply. If so, p_i does it on behave on p_j and then S gets the value $(s', i+1 \mod n)$. This is just the high level description, consider issues in the actual implementation. Mainly: how do I know if someone else is doing my operation, and if the operation has a returned value how do I know what the value was?

Solution is hold have another field in the announcement array to hold this sort of information. Unfortunately, writes are now no longer atomic so as a result, not implementation is not wait-free. One option: put a response field in the CAS object. When you are CAS-ing, just add it in the response! Yes, the CAS objects get quite large! (Even worse, you can put the sequence of operations all in one CAS object! But it works, so what the heck... Solution 0.)

Back to our response field idea: the response field might get over written. Before applying an operation, check the response field of the announcement array to see if it has already been performed. Further we need another field to keep the id of the process whose operation was performed. More problems: everyone needs to help write the response of S into the announcement array. If two different operations are doing this one is really slow, it might complete its write after the information has gone stale. The solution here is to an array of CAS objects, and every entry has a sequence number. This works. Solution 1.

Instead of sequence numbers use memory allocation (this is actually pretty common in distributed computing!). So entry of announcement record now is a pointer to a record. Everyone keeps a local copy and CAS the pointer as needed. The problem is the large amount of memory needed. The solution is a reference counter. If you are trying to access $A[i]$, follow its pointer, increment the reference counter, then check again to make sure the reference of $A[i]$ did not change (reference counter needs to be kept locally and cannot be reused). Notice that each process needs n local records. Solution 2.

Instead of round robin helping we can use operation combining. Collect the announce array and locally perform all operations that do not currently have a response. Try to CAS. If you fail do it again. If you fail again, then you can stop since you know the second CAS read your announced information so did the work for you. Solution 3 (not in the textbook). ■

As before, we are going to go through the notes. This time using the book as a reference... see this is the problem if you don't post the notes! There is nothing to fall back on so the students just end up using another, more complete resource.

Claim 8.2 CAS *is a universal object, but we will only show a non-blocking simulation here.*

Proof: Before we begin, we note some limitations of the construction (yes, very optimistic we know). First, as stated in the claim, the simulation is not wait free. Next it uses unbounded memory for both the records and sequence numbers. Finally CAS objects are not that general, we would prefer it if we could use general consensus objects. All this will be fixed in the next section, but for the time being lets do this.

The big idea for the simulation is as follows: we will take a sequence of operations and string them together in a linked list. There will be a CAS object $HEAD$ controlling which process gets to add their operation to the list by controlling the pointer to the first operation. Records are assembled locally by each process which then tries to set $HEAD$ to be this newly created record.

Records are of the form $\langle inv, new - state, response, before \rangle$ which represents the operation's invocation parameters, the new state of the object after the operation's execution, responses of the operation, and a pointer to the previous operation. Note that in this simulation the current record points to a previous one. Initially $HEAD$ points to an anchor record with values $\langle \perp, \mathit{inital}, \perp, \perp \rangle$.

This simulation is not wait-free since it is possible that one process gets to perform all its operations. ■

Claim 8.3 *Consenus is universal in a system of n processes.*

Proof: Use the consensus object like the CAS to choose the sequence of operations. This produces a linked list of operations where each record has a data, state and next field (you would apply the consensus object to the next field to determine the next state). ■

Claim 8.4 *Consensus is universal in a system of n processes (again).*

Proof: We will build on the solution using CAS above, but since a consensus object are one time use, we will change the $before$ entry in our record into an $after$ record. Now the chain of records will point from old records to more recent ones. This causes some issues when we want to find the head of the list. Thus we will keeps a public array where every process will write down a pointer to the latest record they saw at the head of the list along with a sequence number. Every time a process wants to perform a new operation, it can just check the array and choose the pointer with the largest sequence number.

Unfortunately, theses ideas are still insufficient for a wait-free simulation. For that we need the idea of *helping* (in particular *round robin helping*). What we want is for every process to get a turn at putting their operation into the list if indeed they have operations to add. We do this by maintaining a $\mod n$ counter. When the counter is at i then every process tries to process p_i's operation into the list. In order to do this, we will need to maintain an *Announce* array where every entry is a pointer to a record that a process wants added to the list. Initially all pointers in *Announce* point to the anchor record. When a process p_i wants to add a record to the list, it sets up the record and changes its pointer in *Announce* to point to this record. Then p_i looks at the current index in the counter. Suppose the index is j. p_i will see if $Announce[j]$ points to any record and will help add it to the list. Eventually the index of the counter will become i. Before trying to add its record to the list, p_i will see if the record has been assigned a sequence number (this means that some other process already added the record to the list). ■

8.2 Simulating Message Passing in Shared Memory Systems (AW 5.3.3)

This is the easy one. To simulate message passing each pair of node has two registers (arbitrarily large for asynchronous communication) representing the messages sent in either direction.

8.3 Simulating Shared Memory in a Complete Message Passing Systems (AW 10.4)

Suppose we have n processes in an asynchronous system and at most $f < \frac{n}{2}$ crash failures. This is the more interesting one. There is a lower bound result as follows.

Lemma 8.5 *If there was $f \geq \frac{n}{2}$ crashes then this is impossible.*

Proof: Divide the set of all processes into two group A and B of size $\frac{n}{2}$ each. Initially the contents of the simulated registers is 0. Suppose p_0 in A writes a 1. At some time t_0, this write operation must finish. Suppose at time $t_0 + 1$ process $p_1 \in B$ begins a read operation. We will make messages passing between A and B really slow. Thus p_1 does not hear anything from A. Since $f \geq \frac{n}{2}$ processes can fail, p_1 is unsure if all the processes have failed or are just slow so must output 0 for the read operation at some time t_1. We can now let messages through after this time. Observe that the write and read operations are not linearizable since the write occured before the read but p_1 still read a 0. ■

The some ideas on how we would do this: (ABD construction)

Proof of Correctness. Use sequence number to linearize processes. The sequence number associated with $WRITE$ is during the first broadcast. The sequence number associated with $READ$ is during its write.

Algorithm 1 Simulating Shared Memory in a Complete Message Passing System

```
1: $\rightarrow WRITE(v)$
2: broadcast $v$.
3: $\rightarrow READ()$
4: return last value returned from the writer
5: # Problem: NOT linearizable.
6:
7: # New idea:
8: $\rightarrow WRITE(v)$ (has local sequence number)
9: $s \leftarrow s + 1$
10: $broadcast(v, s)$
11: wait for $ack(s)$ from $\geq \lfloor n/2 \rfloor$ processes
12: $\rightarrow READ()$ by $p_i$
13: broadcast a read request $(i, j)$
14: wait until it receives $ack(i, j, v_k, s_k)$ from $\geq \lfloor n/2 \rfloor$ processes
15: compute $k' = \max\{s_k\}$
16: find $k'$ such that $s_{k'} \geq s_k$ for all $k$
17: if $s'_k > s$ then
18:     $s \leftarrow s_{k'}$
19:     $v \leftarrow v_k$
20: end if
21: send $WRITE(v, s)$ to all processes and wait for at least $\lfloor n/2 \rfloor$ acknowledgement messages
22: if process receives $read(i, j)$ then
23:     send $ack(i, j, v, s)$
24: end if
```