Lecture Notes for CSCI 379: Distributed Computing
Set 7.1-ABD Register Implementations

Professor Talmage

April 17, 2023

1 Problem and Model

In an asynchronous, crash-prone (less than half: $f < n/2$), point-to-point message-passing system, we want to implement a SWMR register. First, the paper implements the register using messages of arbitrary size, then refines the solution to use bounded messages. Finally, they implement registers in an alternate communication system where links may come and go (we will not discuss this version in detail).

Note that the paper talks about an *atomic* register. In registers, this is an equivalent condition to linearizability, but atomicity is not necessarily defined for other ADTs. So, while the definition given looks more like regularity, you can use what you know about linearizability.

Definition 1. An atomic register satisfies the following two properties:

- Every *Read* instance returns the argument of the latest completed $Write$ instance or that of a concurrent $Write$ instance.
- If two *Read* instances do not overlap in real time, the value the later instance returns cannot have been written before that returned by the first instance.

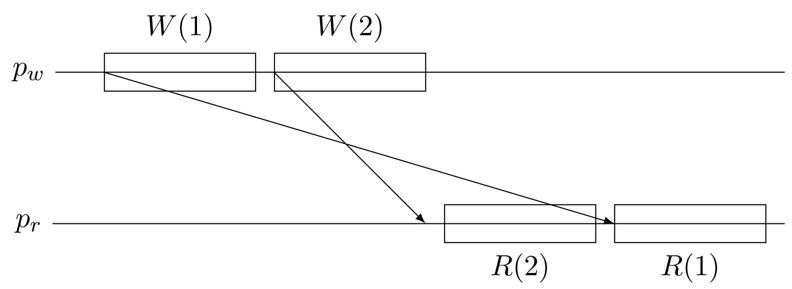
Exercise: How can we speak of "earlier" and "later" $Write$ instances? Convince yourself that this definition is equivalent to linearizability.

This is a *Single-*$Writer$ register, so $Write$ instances are inherently ordered at the writing process. As we've discussed, if we have a SWMR register, then we can implement a MWMR register, so this paper implies a simulation of MWMR registers in a message-passing system.

2 Communicate: Round-Trip Messages and Quorums

First, the paper builds a communication procedure with specific properties. This procedure will make implementing the register functions, and then analyzing them, straightforward. There are two fundamental aspects of this communication procedure to which we should pay attention: acknowledgements and quorums.

First, to guarantee linearizability, we will not want our operations (particularly $Write$) to return until they are sure other processes are aware of them. If they just sent messages and returned, those messages could be very delayed, such that another, non-overlapping operation instance could have invocation, return, message arrival, and effect before this one's messages arrived and it took effect. A later *Read* would then return a stale value, and there would be no legal linearization.



Since we are in an asynchronous system, we cannot predict how long it will take messages to arrive, so instead we rely on acknowledgements. That is, whenever we communicate with the sytem, we wait for acknowledgements from receiving processes. While this may take arbitrarily long, it is only twice the maximum message delay. Which is twice as long as we need to wait, but probably optimal when we do not know the maximum message delay.

The issue with this, of course, is that if a process crashes, we could be stuck waiting for an acknowledgement forever, which is no good. This is the same problem with tolerating crashes in asynchronous systems that has been haunting us since FLP. But here, there is a solution. Because we do not need to ensure all processes are in agreement before we conclude our current operation, we can ignore crashed processes. Thus, when we have received acknowledgements from at least $n - f$ (more than half) of the processes, we can continue and not be stuck waiting for crashed processes.

While the word never appears in the paper, this is an early example of a *quorum system*, which is a fundamental tool for tolerating crashes. We may continue before some correct processes receive our message and acknowledge, but we can be confident that the message we sent is "sufficiently" disseminated. In any future communication, when the sender waits for responses from a majority of the processes in the system, at least one process whose response it gets must have received the first communication, and can include that information in its acknowledgement. There are fancier ways to define quorum systems, since the only necessary property is that each subset (quorum) overlaps with every other, but *majority quorums* as used here are by far the simplest and most common.

Finally, it is worth noting that they define a protocol for using communication links that explicitly enforces FIFO reception of messages over a particular message channel, so the example above is already impossible, before all the stronger guarantees of quorums.

3 Register Implementation

Now we can consider the actual register implementation. To start, I'll give informal pseudocode for the general idea. Note that the paper leaves out the values and *Read* returns, just focusing on choosing the latest timestamp. (Note that in this algorithm, timestamps are just sequence numbers, and I use those terms interchangeably here.)

Exercise:

- What are the complexities of *Read* and *Write*?
- Argue that non-overlapping *Read* instances return values in the order in which they were written.

Algorithm 1 Informal pseudocode for intuition of the unbounded register implementation. Except for $Write()$, code for p_i.

```
1: function READ
2:     Send $R_1$ to all, wait for responses containing timestamps.   ▷ Get values from other readers
3:     Set local timestamp to largest received, local value correspondingly.
4:     Send $R_2$ to all, attaching largest timestamp.   ▷ Reader writes
5:     Wait for responses to ensure ordering.
6:     return local value
7: end function
8: function WRITE($x$)
9:     $seqNum++$
10:    Send $seqNum$ and $x$ to all, wait for responses to ensure ordering.
11: end function
12: Upon event: RECEIVED $\langle W, ts_w \rangle$ FROM $p_w$
13:    If new timestamp larger than local timestamp, update local value and timestamp to those received.
14: end Upon event:
15: Upon event: RECEIVED $R_1$ FROM $p_j$
16:    Send $ACK$ and local timestamp/value pair to $p_j$
17: end Upon event:
18: Upon event: RECEIVED $R_2, ts_j, x_j$ FROM $p_j$
19:    Update local timestamp and value if $ts_j$ is larger
20:    Send $ACK$ to $p_j$
21: end Upon event:
```

4 Bounding Timestamps

Since sequence numbers can increase without bound, running this register implementation for any significant computation will result in unusably large message sizes. To get around this, the paper implements a bounded timestamp system extant in the literature. The idea is that each process tracks the set of timestamps which are currently in use, and can then obtain a new timestamp from a finite domain that is larger than any currently-used timestamp. In the code, this is achievable by overloading the `max` and `++` commands. The complexity comes from saving each new timestamp seen, and discarding timestamps no longer in use.

Idea:

- $Write$ now needs two rounds of communication: one to collect active timestamps and one to send the new value and timestamp (which is larger than any currently active).
- Since the writer can only count on hearing back from a majority of processes, the set of active timestamps must be maintained by quorum.
- When a process learns of a new, larger timestamp, it sends it to all processes and waits for majority acknowledgement.
- Each process stores an $n \times n \times 2$ array of all the timestamps it hears about.
- Would need to read the referenced bounded timestamp paper for details, but only need $O(n)$ bits to store bounded timestamps.

Note that this is actually only better than unbounded timestamps (except for predictability of message size), when there are $o(2^n)$ $Write$ instances in your execution.