Concurrent Algorithms and Data Structures (Sheet #3)

Marius Gavrilescu

1. This protocol achieves consensus but it is not wait-free, because the two "losers" need to wait for the "winners" to agree on a value. The "winners" may be paused for an arbitrarily long time, so the "losers" can wait for any amount of time. Therefore this protocol is not wait-free, so it does not contradict the result we showed in the lectures.

3. Suppose we can implement consensus. Then the protocol will have a critical state. We will look at the next moves of A and B. We know (by the proof for read-write registers) that these moves cannot be reads/writes to a registers or operations on different registers, so both threads must perform a compareAndSet on the same register: this register must be R_{AB}, because it is the only compareAndSet register accessible to both threads. After this operation, we must be in either a 1-valent or 0-valent state (this is determined by the value in R_{AB}). So if we stall A and B and let C run freely, it should arrive at either 0 or 1 (depending on what is in R_{AB}). But C cannot read R_{AB} and there is no other way for C to figure out which thread succeeded in its compareAndSet. So C will act the same whether we are in a 0-valent or 1-valent state, which is incorrect. So we cannot implement consensus in this case.

   With double compareAndSwap we can implement the following protocol: We assume the three shared registers have the same known initial value that is different from the threadID of any of the three threads. Each thread writes their own submission into their own read-write register, then each thread tries to doubleCompareAndSwap with the initial values of the registers (twice) and with the threadID (twice). Clearly only one thread will succeed, that thread can return the number in its own read-write register as the consensus value. The two threads that failed can look at the two registers available to them; one of them will have the initial value, and the other will have the threadID of the thread that succeeded. These threads can then read that thread's read-write register and return that number as the consensus value.

4. For Type A broadcasts it is enough for every thread to broadcast its suggestion, then return the number in the first message received. We know all threads receive all messages in the same order, so the first message received by any thread will be the same (meaning all threads will return the same value), and this message was sent by one of the threads (so the value we agree on was previously submitted by one of the threads).

   A potential algorithm for Type B broadcasts would be for every thread to broadcast its suggestion, then receive all the messages, and finally select the smallest number received. Since every thread receives every message (but not in the same order, because local messages are fast-tracked), every thread will obtain the same smallest number. So all threads will return the same consensus value, and the algorithm is valid (the returned value is the smallest value submitted by any thread, which is clearly one of the values submitted by a thread).

10. Assume we lock only pred. `add` calls `find` and obtains pred and curr, with a lock on pred (but not curr). It then makes a new node with `node.next=curr`, and sets `pred.next=node`, and finally unlocks pred. We need to ensure that between the lock and the unlock no other thread can remove pred, remove curr, or add a node in between pred and curr. We will treat these cases separately.

    Note that `remove` locks both the node it is removing and its predecessor. So a second is unable to remove either curr or pred while the first thread has a lock on pred.

    Now assume a second thread is trying to add a node in between pred and curr. This is the same operation the first thread is doing (but with a different new node). This second thread will also have to acquire a lock on pred before it can proceed, and the lock is held by the first thread. So no other nodes can be added between pred and curr before we unlock.

    So the `add` operation still works in the "optimistic locking" case if we only lock pred and not curr. It is still required that we lock both pred and curr in `remove`