Lecture Notes for CSCI 351: Distributed Computing

Set 4.1-Wait-Free Synchronization[1]

Professor Talmage

February 27, 2023

---

1 The Problem

This paper address the question "What are strong and weak ADT operations?". There may be many aspects to this, so we must establish what it means for an ADT to be strong.

> **Exercise:** Can you think of any features that would suggest that an ADT, or a particular operation, is strong?

One possible answer is that if one ADT can implement another, then it is at least as strong. Recall that we saw this with RMW mimicking $Test\&Set$. If an ADT cannot implement another, then it may be weaker, or simply incomparable. This paper builds a framework to determine the answer to the question "Is it possible to implement ADT X using ADT Y?". In fact, it goes beyond just that question and constructs an infinite integer scale of different strengths of ADTs. These are called *consensus numbers*. Every ADT has exactly one consensus number, and any ADT of consensus number c can implement any other ADT of consensus number $\leq c$. This follows from the *universality* of consensus, which [1] also proves: the paper expresses Consensus as an ADT and shows that it can implement any other ADT.

The typical practical application of this paper is proving the consensus number of a particular ADT. A consensus number proof comprises two parts: First, we must show that we can implement consensus using the ADT in a system with n processes. We must then prove that it is impossible to implement consensus using the ADT in a system with $n + 1$ processes. Together, this tells us that it has consensus number n.

2 Technicalities

There are a few important notes for the model used in this paper.

First, we are considering *wait-free* implementations. The name comes from the fact that we do not want any process to ever be waiting for an action by another process. This is because that other process might crash, in which case we would be stuck waiting forever. Formally, this is an asynchronous, crash-prone shared memory system with $f = n - 1$. So, there are no bounds on how long a process may take to act, and any number of processes may fail. Thus, there is no way to discern a crashed process from a slow one, and we cannot rely on a majority of correct processes to determine what to do. One good thing is that, since this is a shared memory system, there is no notion of a crashing process performing part of its final action. Each shared memory operation is atomic, so it either happens or does not.

Second, we must consider Consensus as an ADT. This is actually not too difficult.

> **Exercise:** Give an ADT specification for Consensus.

Definition 1. The (binary) Consensus ADT offers a single operation: $Decide(x, d)$ which takes one input value from $\{0, 1\}$ and returns one output value from $\{0, 1\}$. A sequence of *Decide* instances is legal iff they all return the same value and if all inputs were equal, the return values are that same value.

- [1] uses the stronger validity condition that the decision value is necessarily some process' input, but that is equivalent in binary consensus.

Third, when we are trying to implement Consensus using an ADT A, we may use any number of $Read/Write$ registers, as well as an object of type A.

3 Simple Registers

First, we will show that $Read/Write$ registers are not strong enough to solve consensus by themselves. This proof will be very reminiscent of the impossibility proofs we studied last week, but there are some important differences. For instance, since we are in a shared memory system, steps are either shared memory operations or local computation.

We start with some definitions and lemmas that will focus our attention:

Definition 2. A configuration is *critical* if it is bivalent, but all child configurations (those reachable from it in a single step) are univalent.

Lemma 1. *Any consensus algorithm has an initial bivalent configuration and must have a reachable critical configuration in every execution.*

Lemma 2. *Every critical configuration has child configurations with different valencies reached by different processes acting on the same object.*

Exercise: Prove these lemmas.

- The argument that there is an initial bivalent configuration is the same as we used last week.
- There must be a reachable critical configuration, or we can always progress to a bivalent child, running forever without deciding, and thus not solving Consensus.
- We can prove the second lemma by noting that local operations and operations on different shared objects can be reordered without changing the resulting configuration. The operations must be by different processes, as each process is deterministic, so it will take the same next step.

Now, we can proceed to prove the consensus number of $Read/Write$ registers.

Theorem 1. *$Read/Write$ registers have consensus number 1.*

Proof. Assume that there is an algorithm A which solves binary consensus between two processes using only $Read/Write$ shared objects.

Let C be a critical configuration reached by running A from an initial bivalent configuration. (Exists by preceding lemmas.) Now, p_0 and p_1 are each ready to invoke a shared operation on some registers R, and their next steps will leave the system univalent, but the valency depends on which is next to act. WLOG, assume that $p_0(C)$, the result of p_0 taking a step from configuration C is 0-valent and $p_1(C)$ is 1-valent. We now consider cases for whether the processes are about to *Read* or $Write$.

1. Suppose at least one process (WLOG, p_0) is about to *Read*. Then $p_0(C) \sim_{p_1} C$, as the only difference is in the local state of p_0. Then, if p_1 runs alone from either C or $p_0(C)$, it will decide the same value. But this contradicts the fact that $p_0(C)$ and $p_1(C)$ have opposite valency.

2. If both processes are about to $Write$ to R, consider what happens when p_0 acts from C vs. acting from $p_1(C)$. We know these configurations have different valencies, so processes must reach different decision values. But in $p_0(p_1(C))$, p_0's $Write$ operation overwrote the value that p_1 wrote, meaning that $p_0(C) \sim_{p_0} p_0(p_1(C))$, and thus p_0 decides the same value in both if left to run alone. Again, the only difference between the two configurations is in p_1's local state, which is lost if p_1 crashes or is delayed.

□

4 Compare and Swap

At the other end of the hierarchy is an object known as *Compare&Swap*. This is much like an *RMW* operation, but slightly more restricted. Instead of applying an arbitrary function to the register's value, *CAS* takes two inputs: an expected previous value and a desired new value. CAS checks to see if the register holds the correct previous value and only updates it to the new value if it does. It returns whatever value was previously stored.

Algorithm 1 (Sequential) Pseudocode for *Compare&Swap* register, with current value V

```
1: function Compare&Swap(old, new)
2:     prev = V
3:     if V == old then
4:         V = new
5:     end if
6:     return prev
7: end function
```

Theorem 2. *A register with Compare&Swap has consensus number* ∞.

> **Exercise:** What do we need to prove this theorem? Give such an algorithm. Outline its correctness proof.

Proof. We merely need to give an algorithm that works for any number of processes n. This algorithm takes advantage of *Compare&Swap*'s semantics and setting an initial value to ensure that only the first process complete *Compare&Swap* stores its input value, and all other processes see and decide that value.

Algorithm 2 Algorithm solving consensus for any $n > 0$ using a *Compare&Swap* object V, initially storing $\perp$. Code for p_i.

```
1: function DECIDE(input)
2:     x = V.Compare&Swap(⊥, input)
3:     if x == ⊥ then
4:         return input
5:     else return x
6:     end if
7: end function
```

□

5 Consensus Number 2

> **Exercise:** Try to determine the consensus numbers of FIFO queues (not augmented–no *Peek* operation). See if you can find an n for which you can solve consensus, or an n for which you can argue that it is impossible.

Theorem 3. *A FIFO queue has consensus number 2.*

Proof. For this claim, we will actually need two proofs. First, we must prove that it is possible to solve Consensus in a system of two processes using a queue. Second, we need to prove that it is not possible in a system with 3 processes.

Algorithm 3 Algorithm to wait-free solve Consensus among $n = 2$ processes using a shared FIFO queue Q. Code for p_i.

```
    Initially, Q contains one element, ⊤
1:  R_0, R_1 are shared Read/Write registers
2:  function DECIDE(input)
3:      R_i.Write(input)
4:      x = Q.Dequeue()
5:      if x == ⊤ then
6:          return R_i.Read()                    ▷ Or simply return input
7:      else return R_{1-i}.Read()
8:      end if
9:  end function
```

Correctness

- Agreement: Whichever process is first to *Dequeue* will see $\top$, and the other will see whatever signal the queue gives for empty (often $\bot$). The process which sees $\top$ will decide its own input, and the second process will decide the input of the first. That input is guaranteed to be available, since the first process cannot *Dequeue* until it finishes writing its input. Thus, they decide the same value.
- Termination: Each process performs three operations on shared objects, and never has to wait for any other action, so will terminate in finite time.
- Validity: Processes decide the input of one of the participating processes, so if all inputs are equal the decision value will be that value.

Next, we must prove that it is impossible to solve consensus using a Queue with $n = 3$. Assume in contradiction that some algorithm A does so. As for the proof of the consensus number of a *Read/Write* register, A must have an initial bivalent configuration, which must have a critical descendant C. Suppose, WLOG, that if p_0 acts next, we reach a 0-valent configuration and if p_1 acts next, we reach a 1-valent configuration. Consider the possibilities for processes' next operations (recall that they must be on the same shared object):

- Processes are about to act on a *Read/Write* register. Then the arguments from the proof that such registers have consensus number 1 are valid, and we have a contradiction in all cases.
- Both p_0 and p_1 are about to *Dequeue*: $p_1(p_0(C)) \sim_{p_2} p_0(p_1(C))$, since both p_0 and p_1 are removing elements, the only difference is which process gets which of the top two elements. In either case, p_2's local state is the same, and the states of all shared objects are the same, so if p_2 runs alone, it must reach the same decision from both configurations, contradicting their differing valencies.
- One of p_0 and p_1 is enqueuing (WLOG p_0), the other is dequeueing. First, if the queue is non-empty in C, then the order of the two operation instances doesn't matter and the shared object will end in the same state, leading to a contradiction. If the queue is empty in C, then $p_1(C) \sim_{p_2} p_1(p_0(C))$, since the queue ends up empty in all cases. Contradiction.

- Both p_0 and p_1 are about to *Enqueue*. The only difference between $p_0(p_1(C))$ and $p_1(p_0(C))$ is the order of the last two items in the queue, so any deciding execution must dequeue at least one of these two distinguish between the two configurations of differing valency. Suppose p_0 enqueues x_0 first, then p_1 enqueues x_1. Let p_0 run alone until it dequeues x_0, then let p_1 run alone until it dequeues x_1. Call the resulting configuration C_0. Now, suppose that p_1 had enqueued first. Again, let p_0 run alone until it dequeues x_1, then let p_1 run alone until it dequeues x_0. Call the resulting configuration C_1. The only difference between C_0 and C_1 is which of p_0 and p_1 has x_0 or x_1. Thus, $C_0 \sim_{p_2} C_1$, but C_0 is 0-valent and C_1 is 1-valent. Contradiction.

Thus, in every case, A will generate an decision contrary to its valency, and cannot be a correct consensus algorithm. □

The proofs of the following are left as exercises for the reader.

Theorem 4. *A LIFO stack has consensus number 2.*

Theorem 5. *An augmented queue or stack has consensus number ∞.*

6 Arbitrary Consensus Numbers

To show that the hierarchy of consensus numbers is full–that there are ADTs with consensus numbers other than 1, 2, and ∞, the paper constructs a parameterized type with consensus number $2m-2$, for any integer m. This isn't quite enough to show that there is a type with every possible consensus number, but is close enough to show that the whole space of integers is interesting. (Other work has shown that for every positive integer, there is an ADT with that consensus number.)

The type given in this paper is a m-register assignment operation. That is, a process can atomically write new values to m different registers. m processes can solve consensus using this ADT as follows[1]: Each process has an "announcement" register, much like that in the queue algorithm above, and every pair of processes shares a register. Each process writes to its announcement register and the $m-1$ registers it shares with other processes in one step. By reading each of the two-process registers, we can then determine the relative ordering of when any pair of processes completed their m-way $Write$ operations (see if one or both hasn't finished, if both finished, the value in their two-process register is that of the later process). Combining these relative orders, each process can determine which was first overall and decide the value in its announcement register.

This bound is then extended to $2m-2$ processes by running that algorithm separately on two halves of the system, then pitting the resulting two decision values against each other.

7 Universality of Consensus

What remains to be done is to show that Consensus is *universal*. That is, if we have a Consensus object, then we can implement any other ADT we want. This is important, as transitivity then tells us that if we have an object of an ADT with consensus number n, we can implement any type we want in a system of n processes. Chaining implementations as that implies is not necessarily the most efficient way to do so, but we know it is possible.

Intuitively, to implement an arbitrary ADT using consensus, all we need to do is have processes participate in a round of consensus for each operation instance in the system. All processes propose the operation they want to execute next and consensus determines which one is next. Since all processes agree, they all have a consistent view of the state of the implemented object. They then participate in another round of consensus to determine the next operation instance, and so on. Some processes may lose consensus many, many times, but that is allowed in the wait-free (asynchronous, any number of crashes) model.

[1] The paper layout here is confusing–the pseudocode in the middle of this proof is for a different ADT on the previous page.

References

[1] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.