Lecture Notes for CSCI 351: Distributed Computing
Set 4-Consensus

Professor Talmage

February 24, 2023

1 The Problem

The next core distributed computing problem we will consider is the Consensus problem. This is a distillation of the core notion of forging agreement between processes, and thus turns up in all parts of distributed computing. In fact, Leader Election and Mutual Exclusion can be seen as special cases of Consensus (agree on a leader, repeatedly agree who progresses next). As such a fundamental problem, there are many aspects of consensus, many solutions and impossibility results in different models, and many applications. We will consider a few general results and one particular application of consensus which is fun and gives a lot of insight into distributed data structures.

Problem 1. Each process has an input value known only to itself. Each process must return a value, satisfying:

- **Termination**: All correct processes must return in finite time.
- **Agreement**: All correct processes must return the same value.
- **Validity**: If all processes have the same input, they must return that value.

Exercise: Give a simple algorithm excluded by the Validity condition.

- We often use "decide" instead of "return".
- In binary consensus (all inputs/outputs are 0 or 1), Validity is equivalent to saying that processes must decide some process' input. In multi-valued consensus, requiring the decision value to be some process' input is a strictly stronger conditions.

Exercise: Solve Consensus using either Leader Election or Mutual Exclusion. That is, pick a previous problem, assume you have a working solution, and write pseudocode to solve Consensus using that solution.

Exercise: Solve Leader Election or Mutual Exclusion using a black-box solution to Consensus.

1.1 Models

Consensus is interesting in (probably) all models of distributed computation, and we will consider it in several. In particular, we will use this opportunity to start considering failing processes. Recall that failures are typically categorized as one of two types:

- A *crash* failure means that a process stops taking any actions at all. Formally, we extend the definition of admissible executions to allow some processes to stop taking steps. Note that a crashing process may send some, but not all, of the messages the algorithm says to in its last step, and we don't know which.

- A *Byzantine* failure means that a process begins acting arbitrarily. Formally, in each step, it may take any shared action (shared memory operations, send any messages) and move to any local state.

We will use f as a system parameter for the maximum number of processes that may fail in a single execution. We will need to specify for each result what types of failures are possible.

- $0 \leq f < n$, since there cannot be more failures than processes, and if all processes may fail, then nothing is possible.

- Any algorithm (upper bound) which *tolerates* Byzantine failures will also tolerate crash failures, since a Byzantine process may act like a crashed one. Conversely, an impossibility result or lower bound in a system with crashes implies the same result in a system with Byzantine processes.

2 Algorithms

We will first consider a synchronous, message-passing, crash-prone system. Remember that we start with an easier model like this to gain an understanding of the difficulties of the problem, and because any lower bounds and impossibility results will carry over to more difficult models.

2.1 Synchronous, Crash Failures

Exercise: Devise an algorithm which solves consensus in this system. Don't worry about efficiency. Assume the communication graph is fully-connected.

Algorithm 1 Simple Consensus algorithm for a synchronous, crash-prone, message-passing system. Code for p_i with initial value x_i.

```
   Local variable $V = \{x_i\}$
1: for round $1 \leq k \leq f+1$ do
2:     send $\{v \in V \mid p_i$ has not already sent $v\}$ to all other processes
3:     receive $\{S_j\}_j$ from $\{p_j\}_j, 0 \leq j \leq n-1, i \neq j$        ▷ May not receive from all, due to crashes
4:     $V = V \bigcup_j S_j$
5: end for
6: return $min(V)$
```

Analysis

- **Complexity** (implies termination): $f+1$ rounds (synchronous)

- **Validity**: The decision value is some process' input, since that is all that we put in V, so if all inputs are equal, that is also every process' decision value.

- **Agreement**: With $\leq f$ crashes and $f+1$ rounds, there must be at least one round with no failures. In a round with no failures, the entire set of values known to any active process is propagated to all nodes, since all processes send everything they know (except for redundancies). Thus, for any value any process decides, all processes had that value, so the minimum of decided values was known to all, and thus decided by all.

2.2 Byzantine Faults

Aside: Byzantine Generals: The term “Byzantine” comes from an analogy, invented to explain the consensus problem. The Byzantine army is besieging a city. There are different generals commanding different divisions of the army. They need to decide when to attack by sending messengers to each other. The problem is that some generals are traitors and will attempt to confuse the others, to their destruction. They may lie about their intentions, and may say different things to different generals. If all the loyal generals attack together, they will succeed in taking the city. If all decide to remain where they are, the siege will continue. But if some attack and some remain, they will be defeated in detail, leading to a complete loss and the siege breaking.

We now consider the Consensus problem in a synchronous, message-passing, Byzantine-fault prone system.

Exercise: How can we extend our previous ideas to ensure that all processes can agree on the same value, despite Byzantine processes sending different information to different processes?

Idea 1 Instead of just sending all values heard so far, send full reports of which processes sent which values. Thus, messages look like “p_1 said that p_2 said that p_8 started with value 3”, etc. Store all this information in a trie, where process ID controls the branch, and the sequence of IDs from the root to a node indicates the chain of messages. For $f+1$ rounds, send the lowest level of this tree to all and use it to create a new level of the tree, where each message from p_j adds a child to each node in the previous level.

- Once the tree is built, run a recursive majority protocol up from the leaves to find the decision value
- This algorithm requires $n \geq 3f+1$, so there are a super-majority of correct processes, which will give us agreement.
- $f+1$ rounds and $n \geq 3f+1$ are proven optimal (we will look at the round lower bound, at least, later).
- Messages may have (badly) exponential size

Idea 2 (Phase King Algorithm) Assume process IDs are $0, \ldots, n-1$. We will split time into phases, and in each phase k, process p_k is in charge (king). In a phase, the king will collect values and attempt to persuade all other processes to agree on its value. If the king is correct, it will succeed in doing this, and even a malicious king cannot break an existing agreement. In $f+1$ rounds, we are guaranteed to have a correct king, so we will end up with agreement. Forging and maintaining agreement is done by relying on the strong numerical superiority of correct processes–we count agreeing and disagreeing values and only go with a super-majority that must have lots of correct processes.

Complexity

- $2(f+1)$ rounds
- $n > 4f$ (**Not** $\geq$)

  Exercise: Why is this the algorithm’s failure-tolerance?

- Messages: one value each, so constant size. $O(n^2)$ messages per round, so $O(fn^2) = O(n^3)$ total messages.

Neither round complexity nor failure tolerance is optimal, but both are close and we vastly decreased message size.

Algorithm 2 Phase King algorithm to solve Byzantine consensus, code for p_i with initial value x_i.

```
    Local variables $pref[i] = x_i, pref[j] = \perp$ for $j \neq i$
1:  for phase $1 \leq k \leq f+1$ do
    round $2k-1$:
2:      send $pref[i]$ to all
3:      receive $v_j$ from each $p_j, j \neq i$ and save as $pref[j]$
4:      let $maj$ be the majority value of $pref[0..n-1]$ ($\perp$ if no majority)
5:      let $mult$ be the number of times $maj$ appears in $pref$
    round $2k$:
6:      if $i == k$ then send $maj$ to all
7:      end if                                          ▷ Currently king
8:      receive $king\_maj$ from $p_k$                  ▷ $king\_maj = \perp$ if king doesn't send
9:      if $mult > n/2 + f$ then
10:         $pref[i] = maj$
11:     else $pref[i] = king\_maj$
12:     end if
13: end for
14: return $pref[i]$
```

Correctness

Lemma 1. *If all correct processes prefer v at the start of phase k, then they all prefer v at the end of that phase.*

Exercise: Prove this claim from the pseudocode

Proof. $n/2 + f < 3n/4$, so correct processes will not adopt the king's value, instead taking the majority value, which was their previous preference. □

Lemma 2. *If phase-king k is correct, then all correct processes have the same preference at the end of phase k.*

Proof. Consider any pair of correct processes p_i and p_j. There are three cases for their behavior:

Exercise: Given the three possible cases below, argue each one.

1. Both use king's value. Then they agree. (Note that this doesn't hold for a faulty king.)

2. p_i uses king's value, p_j uses majority value (WLOG). For p_j to use majority value, it must have received that value $> n/2 + f$ times, which means that every process received it at least $n/2$ times, and the king's value is the same, so they agree.

3. Both use majority value. The threshold for majority precludes faulty processes creating different majority values, so agree.

□

Validity follows directly from the first lemma. Thus, since there are $f+1$ distinct kings, there must have been a correct king, after which all correct processes prefer the same value, and they continue to do so until they decide that value, proving agreement.

3 Lower Bounds and Impossibility Results

We'll now move to impossibility results. Despite Consensus being fundamental to nearly any type of progress, it is actually a very difficult problem, so much so that it is often impossible to guarantee a solution. We will look at a few of the many results, largely to get a feeling for the relevant techniques.

3.1 Synchronous Round Lower Bound

We'll start in a friendly model: Synchronous and crash-prone. For these results, we will also limit ourselves to binary consensus. If we cannot solve this, then we certainly cannot solve any more general form of the problem.

To start, we need to introduce a couple of definitions that give us the tools to work on the problem. First, we will define similar executions, much like we did for shared memory. This will again allow us to build executions from which processes must act the same way. Our second definition is looking ahead, discussing what eventual decision(s) processes must make to solve consensus. Combining these, we will create similar executions with different required decision values, showing that any algorithm will lead some process to decide an incorrect value.

Recall that a message-passing execution is a sequence of events at different processes, each of which is either computation or message arrival.

Definition 1. Two executions E and F are *similar* w.r.t. process p if $E|_p = F|_p$. Here, $E|_p$ is the restriction of E to events at process p. We denote similar executions as $E \sim_p F$.

Definition 2. A configuration C of a Consensus algorithm is said to be *S-valent* if S is the set of values for which there is an admissible execution fragment starting in C where a correct process decides that value. If $|S| = 1$, then C is *univalent*, if $|S| = 2$, C is *bivalent*

- For general consensus, $|S|$ may be greater than 2.
- $|S| > 0$, or the algorithm cannot solve the problem from that configuration, which means that either the configuration is unreachable or the algorithm is incorrect.

Theorem 1. *Any binary consensus algorithm for n processes which tolerates f crashes requires at least $f + 1$ rounds, assuming $n \geq f + 2$.*

Idea We're considering the worst-case, so assume that we never have two processes crash in the same round.

- There is a bivalent initial configuration.
- For up to $f - 1$ rounds, there is a step which leads to another bivalent configuration.
- From one of these bivalent configurations after $f - 1$ rounds, there is an execution in which some process does not immediately (in round f) decide.
- Thus, any algorithm takes at least $f + 1$ rounds.

Lemma 3. *Any binary consensus algorithm has a bivalent initial configuration.*

Proof. Assume in contradiction that all initial configurations are univalent.

Let $I[0..n-1]$ denote the processes' input values. Of the 2^n possible values for I, consider those that are only 0s then 1s:

- $I_0 = [0..0]$

- $I_1 = [1, 0..0]$
- $I_2 = [1, 1, 0..0]$
- ...
- $I_{n-1} = [1..1, 0]$
- $I_n = [1..1]$

That is, I_a has a 1s, then $n - a$ 0s. This is only $n + 1$ of the 2^n possible lists of input values, but we can show that one of this reduced set must be bivalent.

I_0 must be 0-valent, as Validity requires that processes decide 0 when all inputs are 0. Similarly, I_n must be 1-valent. As we progress down the list (increasing a), we start with a 0-valent configuration and end up at a 1-valent configuration, so there must be some a s.t. I_{a-1} is 0-valent and I_a is 1-valent. Note that these two configurations differ only in the input value of p_a.

Consider executions from configurations I_{a-1} and I_a in which p_a crashes initially. That is, it crashes in the first round, without sending any messages. For any other process $p_j \neq p_a$, $I_{a-1} \sim_{p_j} I_a$. Thus, if we run the algorithm from I_{a-1} with p_a crashed, p_j decides 0, since it started from a 0-valent configuration. Similarly, if we run the algorithm from I_a with p_a crashed, p_j decides 1. But this is a contradiction, as processes must behave identically in executions from similar configurations. Thus, there must have been a bivalent configuration (specifically, either I_{a-1} or I_a.) □

Lemma 4. *For $0 \leq k \leq f - 1$, there is a k-round execution prefix of any binary consensus algorithm A that ends in a bivalent configuration.*

Proof. This is an inductive proof on k, with Lemma 3 as the base case. This is actually a finite inductive proof, as we must crash a process in each round to prove the inductive step, and we can only crash f total processes (and have already crashed one in the base case).

Inductive hypothesis: Assume that there is a $k-1$ round execution E ending in a bivalent configuration C_{k-1} for $k - 1 \geq 0$.

Inductive step: By contradiction. Assume that all 1-round extensions of E yield univalent configurations. We consider a sequence of possible extensions, similar to lemma 3:

1. If no process crashes in round k, say WLOG that the resulting configuration is 1-valent.
2. Since C_{k-1} is bivalent, there is another possible 1-round extension of E yielding a 0-valent configuration. We assumed that at most one process failed, so let p_i be the process which fails in round k. In its crash step, p_i's messages to some set $Q = \{q_1, \ldots, q_m\}$ of processes were not delivered.
3. Consider the various 1-round extensions of E where p_i crashes and different sets of messages were not delivered. Specifically, consider the sequence $Q_0 = \{\}, Q_1 = \{q_1\}, \ldots, Q_m = Q$, where in each execution Q_a, the processes in the set Q_a failed to receive messages from p_i.
4. Q_0 is the same execution as if no process crashed, so it must be 1-valent. We defined Q_m such that the corresponding extension is 0-valent.
5. Thus, there is some t such that Q_{t-1} is 0-valent and Q_t is 1-valent. For all (non-crashed) processes $p_j \neq p_t$, $Q_{t-1} \sim_{p_j} Q_t$, so if p_t crashes in round $k + 1$, all processes will decide the same value in executions extending from Q_{t-1} and Q_t, contradicting their different valencies.

□

Lemma 5. *If an $f - 1$ round execution ends in a bivalent configuration, there is a 1-round extension in which some non-faulty process has not decided.*

Exercise: Prove this lemma, using a similar technique as in the previous two lemmas. Most importantly, count crashes to make sure we can crash a process in this round.

3.2 Asynchronous Impossibility

The next result is one of the most well-known, fundamental, and disappointing results in distributed computing.[1] If we do not have a synchronous system to allow easy detection and mitigation of failures (as opposed to slow processes), then we cannot solve consensus in the presence of any faults. Intuitively, the idea is that processes cannot tell whether a process is slow or crashed, and the Termination requirement will force them to decide a value. The apparently-crashed process can then wake up and decide the other value, violating Agreement.

On its face, this means that distributed computing of nearly any kind is impossible in any real system, where processes fail. Of course, in the real world, we solve consensus all the time. There are probabilistic solutions, which terminate with high probability. Paxos and Raft are probably the most famous pure consensus algorithms, but they have complex assumptions about there being some synchrony in the system. Even more famous these days are blockchain algorithms. We may spend some more time on them later in the semester, but they are fundamentally consensus algorithms, where participants agree on the next "block" in the chain repeatedly. Since all processes must agree, there is a canonical sequence of blocks in the chain, which cannot change. But, of course, if synchrony fails or any participant fails, then at least one of the consensus guarantees may be violated.

The structure of the proof is very similar to that of our round lower bound in the synchronous model. We first show that there is a bivalent initial configuration, then show that every bivalent configuration can lead to another. The trick is that we cannot keep crashing processes, since $f = 1$. Instead, we use asynchrony to delay a process, then wake it up again at an inopportune time.

Lemma 6. *Any consensus algorithm has a bivalent initial configuration.*

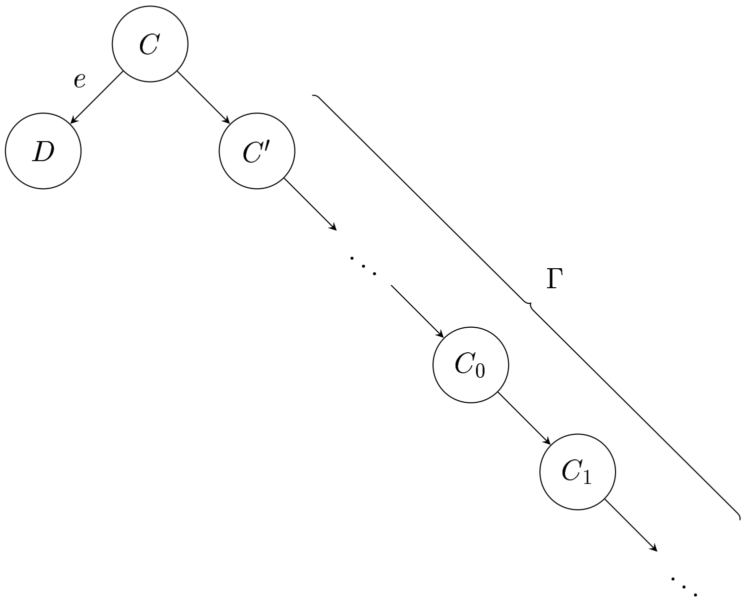
Proof as before.

Lemma 7. *Let C be a bivalent configuration and e a step process p_i may take from C. There is a schedule fragment, ending with e, that yields a bivalent configuration.*

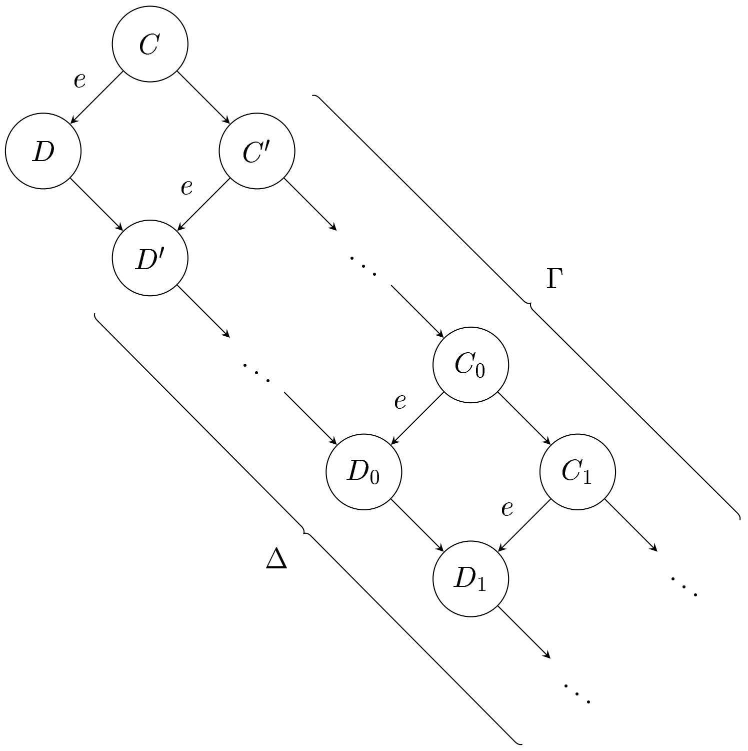
There's another fact, that if we have two schedules from the same configuration, but the sets of processes in the schedules are disjoint, then they can happen one after another, in either order, and yield the same configuration. This follows since we can delay any messages from processes in the first schedule to those in the second until after the second schedule completes.

Proof. First, let Γ be the set of all configurations reachable from C without applying step e.

[1]For further reading: "Impossibility of Distributed Consensus with One Faulty Process" by Fischer, Lynch, Paterson in Journal of the ACM, 1985



Next, consider what happens if we apply e to each of the configurations in Γ. Call the set of resulting configurations Δ.



Note that every configuration in Δ has e as the final preceding step. So, our claim is equivalent to showing that Δ contains a bivalent configuration. Assume in contradiction that it does not.

We first note that Δ must contain both 0-valent and 1-valent configurations. If it did not, suppose WLOG that all configuration in Δ are 0-valent, then there would be a 1-valent configuration in Γ. If we apply e to that configuration, then we have a configuration in Δ that must be 1-valent, as all descendant configurations of a 1-valent configuration are 1-valent.

Next, note that there must be a pair of adjacent configurations in Δ with opposite valencies. Assume WLOG that D_0 is 0-valent and D_1 is 1-valent. We are now ready to restrict our attention to C_0 and its descendants. Call the step that moves the system from C_0 to C_1 e', and p_j the process at which e' occurs.

$$\begin{tikzcd}
& C_0 \arrow[dl, "e"'] \arrow[dr, "e'"] & \\
D_0 \arrow[dr, dashed, "e'"'] & & C_1 \arrow[dl, "e"] \\
& D_1 &
\end{tikzcd}$$

We now have two cases to consider:

1. $p_i \neq p_j$: Then the sets of processes in schedules e and e' are disjoint, so applying them in either order yields the same result. That is, $e'(e(C_0)) = e(e'(C_0)) = D_1$, but then D_1 is a 1-valent descendant of the 0-valent D_0, which is impossible.

2. $p_i = p_j$: Here, what p_i does determines whether the output is 0 or 1, so we pause p_i until some process decides. This must happen in finite time, since the algorithm must decide in finite time, even if one process stops participating (crashes). Call the schedule from C_0 until some process decides σ and the resulting configuration A.

$$\begin{tikzcd}
& & C_0 \arrow[dll, "e"'] \arrow[drr, "e'"] \arrow[ddd, "\sigma"] & & \\
D_0 \arrow[ddd, "\sigma"] & & & & C_1 \arrow[d, "e"'] \\
& & & & D_1 \arrow[dd, "\sigma"] \\
& & A \arrow[dll, dashed, "e"] \arrow[dr, dashed, "e'"] & & \\
A_0 & & & \bigcirc \arrow[r, dashed, "e"] & A_1
\end{tikzcd}$$

Since p_i does not act in σ, applying e and/or e' either before or after σ will result in the same configurations. Thus, A_0 and A_1 are both descendant configurations of A, which implies that A is bivalent. But some process has decided some value x in configuration A, so A is univalent, specifically x-valent, which is a contradiction.

We thus conclude that there is a bivalent configuration in Δ. □

Theorem 2. *For any algorithm A which solves binary consensus in an asynchronous, crash-prone system with $f \geq 1$, there is an infinite admissible execution of A containing only bivalent configurations.*

Proof. We prove the claim by giving an algorithm to construct such an execution.

1. Start from a bivalent initial configuration.

2. Run to a bivalent configuration in which p_0 stepped last. (We just proved that such a configuration exists.)

3. From there, run to a bivalent configuration in which p_1 stepped last.

4. Repeat for each p_i, $2 \leq i < n$.

5. GOTO 2

Since every process takes an infinite number of steps and all messages are eventually delivered, the execution is admissible. □

Corollary 1. *There is no algorithm which solves binary consensus in an asynchronous, crash-prone system.*

Proof. Since there is an infinite, admissible execution consisting of only bivalent configurations, no process decides in that execution, as any configuration in which a process has decided is univalent. Thus, this execution violates the Termination condition of Consensus. □