Lecture Notes for CSCI 351: Distributed Computing
Set 9-Randomized Consensus

Professor Talmage

April 27, 2023

---

1 Introduction and Formalities

Just as in sequential computing, adding randomness to our algorithms can let us do things that are impossible in a deterministic setting. Specifically, we can solve problems we have previously shown to be impossible, we can solve problems more quickly than possible deterministically, or tolerate more faults. We will primarily focus on showing that we can solve asynchronous, fault-tolerant consensus, counter to the FLP result, since consensus is fundamental and necessary for most distributed computations.

In addition to adding randomness, the way we circumvent lower bounds and impossibility results is to change the problem. Of the three conditions we require an algorithm to meet to be considered a solution to the consensus problem, we want to keep Agreement and Validity–processes still need to decide the same value, and that value must make some sense. Termination, however, we will weaken. Instead of guaranteeing that each process will decide a value in finite time, we will allow algorithms to guarantee that processes decide (in finite time) with non-zero probability. In practice, this is nearly the same thing, particularly since algorithms exist (such as that we consider here) which guarantee termination with probability approaching 1.

Formally, we give each process a source of random information which it can use as an input to its transition function at each step. For our purposes, we will assume that this source of randomness is an integer from a fixed range drawn with uniform probability. Different processes' sources of randomness are independent–they cannot assume that another process drew the same number. One of the things we will need to build is a *common coin*, a collective source of randomness.

Exercise: Can you solve Leader Election in an anonymous ring using such a source of randomness?

- Have processes choose IDs randomly, then run a non-anonymous LE algorithm.
- Need to choose a non-anonymous algorithm that tolerates duplicate IDs.
- Alternately, can repeatedly draw new IDs until they are all unique, then take largest.

2 Consensus

This particularly randomized consensus algorithm (there are **many**) has several parts:

1. A communication system: A way for to guarantee $n - f$ processes collectively communicate their values.
2. A common coin: A way for processes to probably get the same random value.
3. A voting protocol: Processes try to converge on a single value. This is the "true" consensus algorithm.

We assume that $n > 2f$, communication is asynchronous message passing, and the communication graph is complete.

2.1 Get-Core

`get-core` is essentially a broadcast function that repeats until it is sure that a large number of values are known by all processes. It goes through three rounds, each collecting values from at least $n - f$ processes, then broadcasting all known values.

- On invocation, basic-broadcast the parameter.
- When receiving a value from p_j, store it at index j in an array of values. When you have received $n - f$ values, basic-broadcast the list with tag 2.
- When receiving a list with tag 2, add any missing values to your values array. When you have received $n - f$ messages tagged 2, basic-broadcast your values list with tag 3.
- Again, when receiving a list with tag 3, add any missing values to your array. When you have received $n - f$ tag-3 messages, return your values list.

A counting argument proves that there is a collection of more than half of the processes whose values are in the lists returned by every process. The runtime, for $n > 2f$, is $O(1)$, since the number of rounds is constant.

2.2 Common Coin

A common coin is a function all processes call that will return the value v with probability, or bias, b. If b is high, then processes are likely to all see the same random value, as if a coin was flipped in the center of the circle where all could see. Of course, there is some probability that any given process will not see value v, and thus processes may see different values. Like a physical coin, common coin functions are typically binary, returning either 0 or 1.

The simplest common coin is for all processes to choose a value from $\{0, 1\}$ with uniform probability.

Exercise: What is the bias of this common coin algorithm? That is, what is the probability that all processes return return value v, for any v?

- The bias is the probability all see a given value v.
- The probability a single process gets 0 is $1/2$, and processes' choices are independent, so the bias is $(1/2)^n$. Same argument for all seeing 1.

We would like to have a bias that is a constant, so that there is actually a reasonable chance of agreement. Constant bias is important, as having that high a probability of agreement on the coin value will limit how many times we expect to have to try getting random values to reach agreement, yielding constant expected running time. There are many common coin algorithms, but we present one.

- Flip a local coin with probability of heads $1 - 1/n$.
- Use `get-core` to share local coin values.
- If any collected coin value is 0, return 0
- Else, return 1

This algorithm gives a bias of $1/4$. To see that the probability of all processes returning 1 is at least $1/4$, observe that $(1 - 1/n)^n \geq 1/4$. To see that the probability of all processes returning 0 is at least $1/4$, we note that `get-core` guarantees that a core set of more than half the processes have their values distributed to all processes. The probability that at least one of these core processes flips a local 0 is more than $1 - (1 - 1/n)^{n/2}$, which is more than $1/4$.

Note that this common coin algorithm tolerates up to half of the processes crashing. This exceeds the failure tolerance of the entire consensus algorithm, so is safe to use.

Note also that the time complexity of this algorithm is $O(1)$, since it is a single call to `get-core`, which has three rounds of communication.

3 Voting Protocol

The idea is similar to previous consensus algorithms we have seen. Each process will announce its preferred value and decide that value if all the votes it sees agree on that value. Then all processes (including those which decided) will announce the value for which they say the most votes in the voting phase. If all outcomes are the same, then processes set their preference to that value. Otherwise, they use the common coin to attempt to get agreement and restart the algorithm.

The idea is that if all processes start a round in agreement, then all processes will decide that value. It is possible for some processes to decide that value in this round, while others do not, if a process disagrees but does not get its message to all processes in time for `get-core`. In that case, though, by the majority agreement of `get-core`, all processes will have the same majority, so in the next round, all processes will start with the same input, which was that early decision value, and decide that value in that round. Thus, even though some processes may decide a round earlier than others, all will decide the same value. If there is no consensus, in the sense that no process saw unanimous agreement, then all processes will fall back to the common coin to try to get agreeing values. This may happen an arbitrary number of times, since the common coin is not perfect, but the high bias of the common coin means that the odds of continuing decrease quickly with the number of flips.

The interesting part of the analysis is the running time, as this is the part that depends on randomness.

Lemma 1. *For any round $r \geq 1$, the probability that all nonfaulty processes decide by round r is at least b, the bias of the common coin.*

Proof. There are two cases. First, if all processes are choosing their preferences by the common coin, there is a $2b$ probability that they all prefer the same value (b each for agreeing on 0 or

1). Second, if some processes see unanimity of majority (not unanimity of preference, leading to decision) in round $r-1$, then there is a probability of b that any processes which did not see that unanimity all get the same value from the common coin. In either case, if all processes enter round r with the same preference, they will agree in that round. □

Theorem 1. *The expected time complexity of the algorithm is $O(T/b)$, where T is the time complexity and b the bias of the common coin algorithm. For the coin we presented, this gives $O(1)$ expected time complexity for consensus.*

This follows from the previous lemma, as we get a geometric sequence $(1-b)^i b$ for the probability of terminating after a particular number of rounds i. The expected value of such a sequence is $1/b$. Multiply this by T time for each round to get total expected time.