





RandomizedConsensus

[FrontPage] [TitleIndex] [WordIndex]

Note: You are looking at a static copy of the former PineWiki site, used for class notes by James Aspnes from 2003 to 2012. Many mathematical formulas are broken, and there are likely to be other bugs as well. These will most likely not be fixed. You may be able to find more up-to-date versions of some of these notes at <http://www.cs.yale.edu/homes/aspnes/#classes>.

For more up-to-date notes see  <http://www.cs.yale.edu/homes/aspnes/classes/465/notes.pdf>.

These notes give a traditional approach to randomized consensus. The approach based on conciliators and adopt-commit objects as discussed in class is described in  this paper.

1. Consensus

The **consensus problem** in distributed computing is to get a collection of n processes to agree on a bit, even though the system is asynchronous (meaning that the timing of events is unpredictable) and some processes may be faulty (meaning that other processes can't wait for them, since they might have crashed). Each process starts with some bit as input, they communicate in some way (typically either by MessagePassing or by writing notes to each other in SharedMemory), and eventually each process decides on an output value, terminating its participation in the protocol. Formally, a **consensus protocol** must satisfy three conditions:

Agreement

All non-faulty processes output the same bit.

Validity

The common output bit is equal to some process's input bit.

Termination

All non-faulty processes eventually decide.

It is known (FischerLynchPaterson, WaitFreeHierarchy) that consensus is difficult in many systems if we assume deterministic processes. **Randomized consensus** is a way to avoid these results by allowing processes to flip coins. The intuition is that the bad executions constructed in the FischerLynchPaterson impossibility result are a tiny fraction of the space of all executions; so if we can make these bad executions improbable, we can solve consensus with high probability. Formalizing this intuition requires changes both to the model (to allow randomization) and the problem statement (to allow consensus protocols to terminate only with probability 1, instead of in all executions).

2. The intuitive story

Imagine two pedestrians trying to pass each other in a hallway, where they need to reach consensus on whether to pass on the left or on the right. Each initially has a preference for which way they want to go (e.g. right for Americans and left for Britons), and from the FLP result we know that if the pedestrians are deterministic and have their timing controlled by an adversary,

each will give up and adopt the other pedestrian's preference *at exactly the same time*. The results are both embarrassing (starvation) and implausible (doesn't happen in real life).

A solution is for one or both of the pedestrians to announce that they are choosing a new preference by flipping a coin. This gives a 50% probability that both will agree with each other, even if only one of the pedestrians flips a coin. Thus after a constant number of coin-flips, both pedestrians agree, and (assuming they can detect this agreement), they will have solved consensus.

This is the basic idea behind randomized consensus: build a framework for detecting agreement, then flip coins to get the agreement. A complication is that with n processes we may have to wait for $\Theta(n^2)$ rounds before all their coin-flips happen to come up the same. So we want some way to build a **shared coin** that usually gives the same answer to all of the processes, and have them flip that instead.

3. Randomization

To add randomization to the model, we allow processes to do internal steps, known as **local coin-flips**, where the state of the process goes from some state q to some new state q' where q' is chosen according to a probability distribution specified by the algorithm. The simplest version of this is when the process just flips a coin that returns heads or tails with equal probability, and we can describe this algorithmically as if the process has access to a coin-flip subroutine (or coin-flip operation) that returns the outcome of the coin. Note that an implicit assumption is that only one process learns the outcome of the coin—this is what makes the coin *local*. A common trick in designing randomized distributed algorithms is to build from these local coins a **global coin** that all processes agree on and that the adversary doesn't have too much influence over. (This is actually a harder problem than solving consensus, and as we will see below if we can solve it we can solve consensus.)

Once we add randomization, we have to revisit the role of the adversary. Without randomization, the adversary was just a universal quantifier over admissible executions. Now we have to think in terms of **adversary strategies**, where the choices made by the adversary may depend on the outcome of coin-flips that occur during the execution of the protocol. An adversary strategy is typically represented as a function that chooses in each global configurations of the protocol which process executes the next operation. Fixing some single adversary strategy removes all the non-probabilistic nondeterminism from the system, leaving a probability distribution over executions that is determined by the probability distribution over sequences of local coin-flips.

Implicit in the definition of an adversary strategy is the assumption that the adversary can't predict future coin-flips (because their outcomes aren't visible in the current configuration). We may choose to impose additional restrictions on what the adversary can see, e.g. by preventing it from observing anything about the state of the protocol (an **oblivious adversary**) or limiting its ability to observe the internal states of processes, values contained in messages, or values written to registers (a **semi-oblivious adversary**). However, in the most general case we will allow the adversary to see (and react to) everything that has happened so far; this gives an **adaptive adversary**, which we will assume by default. (See `TypesOfAdversaries` for slightly more detail.)

4. Randomized termination

We replace the usual termination requirement with:

Randomized termination

With probability 1, every non-faulty process decides.

The agreement and validity conditions stay the same.

Probability 1 does not mean that termination holds in all executions, just that the set of non-terminating executions becomes vanishingly improbable in the limit. This is analogous to the probability that an infinite sequence of coin-flips contains at least one head ; while there exists a sequence that doesn't have this property (all tails), the probability of this bad sequences is equal to $\lim_{n \rightarrow \infty} (1/2)^n = 0$. Similarly we'll show that the sequences of bad coin-flips that prevent consensus require that we get unlucky forever, which doesn't usually happen.

In computing the cost of a protocol that might not terminate for a while, we have to look at expected cost. This could be expected total work, or expected work done by any single process. We can't guarantee any fixed finite bound holds with probability 1, because if we could, we could get a deterministic protocol by running the guaranteed-to-terminate protocol with all local coin-flips returning heads, which contradicts FLP.

5. Consensus with an adaptive adversary

With an adaptive adversary, we can solve randomized consensus in a wait-free shared memory system in expected $\Theta(n^2)$ total operations; this bound is tight (🌐 Hagit Attiya and Keren Censor, Tight bounds for asynchronous randomized consensus, STOC 2007). The Attiya-Censor protocol is a culmination of roughly 25 years of work on the problem; a survey of the history emphasizing wait-free shared-memory results up to 2001 or so can be found in 🌐 James Aspnes, Randomized protocols for asynchronous consensus, Distributed Computing 16(2-3):165-175, 2003. We'll present the Attiya-Censor protocol for building a global coin embedded in an algorithm for turning a global coin into a consensus protocol that was first described by Tushar Chandra 🌐 Polylog randomized wait-free consensus, PODC 1996 for a semi-oblivious adversary model.

5.1. Reduction to shared coin

Basic idea: build two infinitely long arrays `mark[0]` and `mark[1]` of multi-writer bits, where `mark[b][i]` indicates that some process that prefers `b` has gotten to round `i`. In each round, a process looks to see if the processes ahead of it agree with each other, and if so it adopts their common preference; otherwise it flips a shared coin to decide on its new preference. Because slow processes adopt the common values of fast processes, if a fast process looks over its shoulder and sees that nobody has disagreed with it in the last two rounds, it can decide knowing that the slow processes will join it before they get around to flipping any coins.

Here's the actual algorithm:

```
procedure Consensus(input):
  p ← input
  for r ← 1 to ∞:

    mark[p][r] ← true
    if mark[1-p][r+1]:
      # somebody is ahead of us, join them
      p' ← 1-p
    else if mark[1-p][r]:
      # disagreement in our round, run shared coin
      p' ← SharedCoin[r]()
```

```

    else if mark[1-p][r-1]:
        # no disagreement in this round, keep current value
        p' ← p
    else:
        # no disagreement in previous round either, terminate
        return p
    endif

    if mark[p][r+1] = false:
        # abandon our possibly-losing team
        p ← p'
    endif
end
end
end

```

The proof of validity comes from observing that if nobody has input p , then nobody ever marks a bit in $\text{mark}[p]$, and everybody decides $1-p$ after two rounds.


The proof of agreement comes from carefully analyzing the behavior of slow processes once some process terminates; the essential idea is that once I read $\text{mark}[1-p][r-1] = 0$ after writing $\text{mark}[p][r] = 1$, then any process that comes later either (a) already agrees with me, or (b) hasn't written to $\text{mark}[1-p][r-1]$ yet, in which case it reads $\text{mark}[p][r] = 1$ and $\text{mark}[1-p][r] = 0$ and switches its preference before it reaches round r . So every process enters round r with preference p , and in they all decide p in round $r+1$ at the latest.



For termination, we need to know that the SharedCoin protocol returns each value 0 or 1 with probability at least δ for some constant δ (called the **agreement parameter**), no matter what the adversary does. The reason for this is that in a round where some processes execute SharedCoin, there may be a few fast process that don't execute SharedCoin because they only saw one value in round r . So we need a constant probability that the coin-flippers agree with their fixed value no matter what the fixed value is. But if our SharedCoin has this property, then there is a constant probability that the coin-flippers agree with the leaders, and we get a constant probability per round of termination, with an expected time to termination of $O(1/\delta)$ asynchronous rounds.

5.2. Building a shared coin

Basic idea:

- Each process generates random ± 1 votes and adds them to a common pool.
- Because the adversary can stop processes carrying votes it doesn't like, there can be a difference of up to n between the total generated vote and the total vote written to the registers or counter implementing the pool.
- If the total generated vote S is $\gg n$ or $\ll -n$, these hidden votes don't change the sign of the aggregate vote.
- We need $\Omega(n^2)$ votes to get a constant probability that $|S| = \Omega(n)$ (by the Wikipedia: Central limit theorem).
- Some additional complications ensue because (a) we have to implement the pool as a bunch of single-writer registers, and (b) we can't detect immediately when the generated votes pass the $\Theta(n^2)$ threshold, since we only see written votes.

An approach similar to this (terminating when a random walk reached $\pm\Theta(n)$ instead of at $\Theta(n^2)$ total votes) was used for the first polynomial-time randomized consensus protocol of Aspnes and Herlihy  James Aspnes and Maurice Herlihy, Fast randomized consensus using shared memory, J. Alg. 11(3):441–461, September 1990; their algorithm used $O(n^4)$ total register

operations. Terminating at $\Theta(n^2)$ total votes was suggested by Bracha and Rachman  Gabi Bracha and Ophir Rachman, Randomized consensus in expected $O(n^2 \log n)$ operations, WDAG 1990, which produced a dramatic reduction in the overhead of testing termination. The optimal total work of $O(n^2)$ operations was only achieved in 2007 by Attiya and Censor  Hagit Attiya and Keren Censor, Tight bounds for asynchronous randomized consensus, STOC 2007, by augmenting the Bracha-Rachman protocol with a termination bit that ensures that all processes detect termination at roughly the same time. This is the protocol we describe below:


```
count ← 0
sum ← 0
while not done:
    count++
    sum += flip()
    A[i].(count, sum) ← (count, sum)
    if count mod n = 0:
        if  $\sum A[i].count \geq n^2$ :
            done ← true
        return  $\text{sgn}(\sum A[i].sum)$ 
```



Analysis:

- Let S_t be the sum of the first t votes that are generated by calling localCoin (whether they are later written or not). The sequence of values S_0, S_1, \dots forms a random walk, since the adversary can't control what values localCoin returns.
- When $t=n^2$, S_t is approximately normally distributed, by the Wikipedia: Central limit theorem. In particular, for sufficiently large n and any c , $\Pr[S_t \geq cn]$ is bounded below by a constant.
- Now let t' be the number of additional votes that are generated. By examining the code we see $t' \leq n^2$, since each process can only generate n votes before checking the threshold. We can't apply the Wikipedia: Central limit theorem to the extra votes $S_{t'}-S_t$, because the adversary can stop the process early if the extra votes are drifting the way he likes. But we can use the reflection principle to argue that the probability that a random walk ever exceeds b before some time t' is at most twice the probability that it exceeds b exactly at time t' . This gives a constant upper bound on the probability that the extra votes push the total down by more than n .
- Finally we have to take into account the difference of up to $n-1$ between the generated vote and the votes that any individual process sees. (We get $n-1$ by assuming that all processes flip their coins immediately after checking done, so that once done is true, no additional flips occur; if we don't make this assumption, we get $2n-1$, which is still OK.)

Putting this together gives a constant probability that all processes see the same sign for the total vote as obtained from the first n^2 votes. This gives a constant probability that they agree on each possible value ± 1 .


5.3. Improving individual work

One problem with this protocol is that the bound on individual work is equal to the bound on total work: a single process running in isolation will have to generate all $\Theta(n^2)$ votes itself. Because of the lower bound on total work, we can't hope to get individual work below $O(n)$. But we can in fact achieve $\Theta(n)$ individual work by having processes cast increasingly heavy votes over time. The basic idea of doing this goes back to a paper by  Aspnes and Waarts from STOC 1992;


they modified the Bracha-Rachman protocol to get an $O(n \log^2 n)$ bound on individual work using a carefully-chosen weight function. This bound was improved to $O(n \log n)$ by  Aspnes, Attiya, and Censor in PODC 2008—essentially just by adding the Attiya-Censor termination bit to the Aspnes-Waarts protocol—and finally to $O(n)$ by  Aspnes and Censor in SODA 2009.

The weight function used in the Aspnes-Censor paper is not terribly complicated. Each process keeps track of the total variance v it has generated so far, and sets the weight of the next coin-flip to be equal to v if that gives a weight between $1/n$ and $1/\sqrt{n}$, or clamps the weight to the nearer of these two endpoints if it would otherwise lie outside this range. The process checks done variable after every vote and the total variance (equivalent to $\sum A[i].\text{count}$ in the Attiya-Censor protocol) every time its own contribution doubles. It stops if done becomes true or the total variance exceeds 1.

The important property of this algorithm is that the weight of any pending vote by some process is at most a constant times its previous contribution to the total variance; so when we cross the variance threshold, we have a constant bound on the total weight of any hidden votes. This allows essentially the same analysis as for the Attiya-Censor protocol to go through (a minor complication is that we have to use a fancier version of the CLT to deal with having variable numbers of votes with varying weights).

The only other messy bit of this is that each process checks the total variance counter $\Theta(\log n)$ times, which costs more than $O(n)$ work if we implement the counter using collects. So Aspnes-Censor includes an $O(n^{4/5+\epsilon})$ implementation of an approximate counter based on a combination of sampling tricks and expanders, just to eat that extra $\Theta(\log n)$. This makes the whole algorithm pretty horrifying, but there is  later work that shows how to implement a better counter (with $O(\log n)$ -work operations!) that avoids all the nasty statistical tricks.

6. Consensus with an oblivious adversary

Achieving consensus in $\Theta(n^2)$ total work is pretty expensive. If we relax the assumption of an adaptive adversary, it is possible to do much better. The basic idea (which appears in  Chor, Israeli, and Li, Wait-free consensus using asynchronous hardware, SIAM Journal on Computing, 23(4):701–712, 1994 and which, in its original conference publication, precedes most of the results described above) is to use a round-based race but have processes in the leading round individually flip coins to decide when to advance to the next round. If each process advances only with probability $1/(2n)$, then there is a constant probability per round that exactly one process will advance before the others notice it and adopt its preference. Note however that assuming an non-adaptive adversary is critical; if the adversary can detect that a lucky process is about to advance to the next round, it can delay it until some other process with a different preference is also ready to advance. This requires either assuming that the adversary is oblivious, or making a weaker assumption that prevents the adversary from applying this particular strategy (typically that the adversary can't stop the write that announces an advance based on whether the advance is happening or not).

The behavior of a process is to repeatedly execute the following loop:

1. Look at the position of the other processes. If some process is ahead of me, jump to the leading round while adopting the preference of some leading process.
2. Otherwise, if all processes in my current round and the previous round agree with me, advance to the next round and decide.
3. Otherwise, advance to the next round with probability $1/(2n)$.

In the Chor-Israeli-Li paper, the total work for consensus is still $\Theta(n^2)$. This is mostly because each process spends $\Theta(n)$ time scanning registers belonging to the $n-1$ other processes during each pass through the loop. By reducing this overhead (under slightly stronger assumptions about the adversary) the total work can be improved to $O(n \log \log n)$ (📄 Ling Cheung, Randomized wait-free consensus using an atomicity assumption, OPODIS 2005). With some further tweaks, it is possible to get total work all the way down to $O(n)$, with an $O(\log n)$ bound on individual work; see 📄 here for details.

6.1. Proof that we get a unique winner in $O(n)$ attempts

Suppose that the highest round for any process is r . Observe that (a) any process in round $r' < r$ catches up in at most one pass through the loop; (b) the expected number of passes by processes in round r until some process advances is $2n$; and (c) any other process in round r gets at most one chance to advance probabilistically before it notices the new leader and adopts its value. So starting from an arbitrary state, the adversary can force us to do an expected $3n-1$ passes before some process increases the max round ($n-1$ laggards catching up plus $2n$ expected attempts to advance before one is successful). We then have a probability of $1 - (1 - 1/(2n))^{n-1} \rightarrow 1 - e^{-1/2}$ that no subsequent attempt to advance by a process with a different preference succeeds. If this event occurs, then we are in a state where there is only one leading process; the other processes eventually adopt its preference ($n-1$ passes), some process advances to the following round with the same common preference (expected $2n$ passes), and then all processes advance to that round and decide ($2n-1$ passes). If the event fails, we try again. The total is $\Theta(n)$ expected passes through the loop, which must be multiplied by the cost of each pass, which depends on how the first step of checking the state of the other processes is implemented.

CategoryDistributedComputingNotes CategoryRandomizedAlgorithmsNotes

2014-06-17 11:58