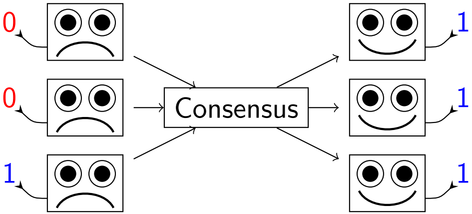
Message-Efficient Randomized Consensus

Dan Alistarh, James Aspnes, Valerie King, and Jared Saia

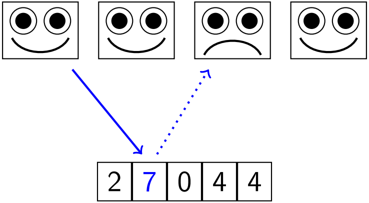
DISC 2014 / 13 October 2014

Consensus



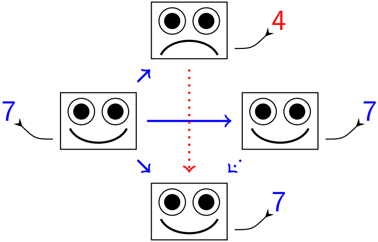
- Consensus (Pease, Shostak, Lamport, 1980) requires all process to agree on the input to some process.
- Want to solve in an asynchronous message-passing with $f < n/2$ crash failures.
- Known to be *impossible* deterministically with even *one* failure (Fischer, Lynch, Paterson, 1985).
- But can be solved with randomization (Ben-Or 1985).
- How to minimize number of messages?

Shared memory version



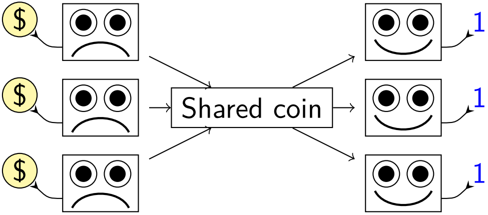
- Randomized consensus well-understood in **shared memory**.
- Expected $\Theta(n^2)$ memory operations *necessary* and *sufficient* for $f = n - 1$ crash failures with **adaptive adversary** (Attiya and Censor, 2008).
- Each process can do expected $O(n)$ operations (Aspnes and Censor, 2010).

Conversion to message passing



- Use standard simulation of (Attiya, Bar-Noy, Dolev, 1995).
  - Write operation: send new value to a majority of processes.
  - Read operation
    - Solicit most recent value from a majority.
    - Send this back to a majority to ensure linearizability.
- Cost: $\Theta(n)$ messages per operation.
  - $\Rightarrow \Theta(n^3)$ expected total messages.
  - $\Rightarrow \Theta(n^2)$ expected messages per process.
- Our goal: per-process cost close to trivial $\Omega(n)$ lower bound.

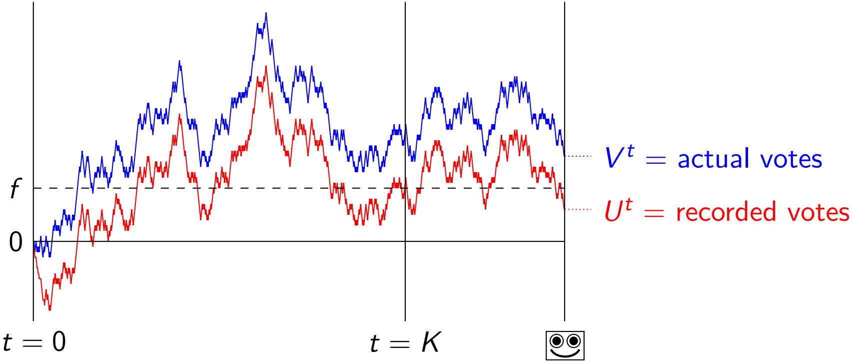
Shared coins



Reduce randomized consensus to a **shared coin**. (Ben-Or, 1985)

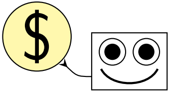
- Each process can repeatedly flip a **local coin** not visible to other processes.
- Want to combine these local coins into a single shared coin.
- Adversary can stop a process before it propagates its local coin.
- Adversary wins if it can get control of the shared coin.
- $\Rightarrow$ retry until adversary loses.

Shared coins by voting (Bracha and Rachman, 1991)



- Generate sum V^t of $t \geq K = \Omega(f^2)$ independent ± 1 votes.
- Adversary can hide at most f votes by crashing processes.
- $\Rightarrow$ *observed* sum U^t satisfies $|U^t - V^t| \leq f$.
- So if V^t exceeds f at $t = K$ and *stays* above f until process p looks at U, then p sees majority > 0.

Impatient voting (Aspnes and Waarts, 1996)



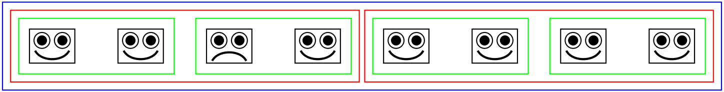
- One process might generate all n^2 votes!
- Solution: have processes cast bigger votes over time.
- One process can generate n^2 *variance* in $O(n)$ votes.
- Handful of fast processes don't give adversary (much) more power.

How do I preserve a vote in message-passing?

1. Send it to all processes.
   - $\Theta(n)$ messages.
   - Adversary can't hide vote once majority receive it.
2. Send it to one other process
   - 1 message.
   - Adversary can hide vote but must crash two processes.
   - But what about subsequent votes?

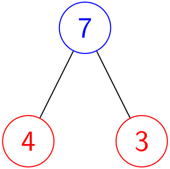
The big idea: Send **big** piles of votes to **big** quorums of processes.

Tree of nested quorums



- Every 2^k votes, I propagate them to 2^{k+1} processes.
- Cost: $O(2^k)$ messages for each packet of 2^k votes
  - $= O(1)$ amortized messages per vote per level
  - $= O(\log n)$ amortized messages per vote.
- Lost votes:
  - 2^k unreported votes $\times$ 2^k processes $= 2^{2k}$ votes in subtree.
  - Adversary kills all of them with $\Theta(2^k)$ failures!
  - But sum of these votes is $O(2^k\sqrt{\log n})$ with high probability.
  - $\Rightarrow$ lost votes per failure is still small.

Node implementation



- Each node in the tree is implemented as a **max register** (Aspnes, Attiya, Censor-Hillel, 2012).
- Reading a max register returns the largest value written.
- This solves the **lost update** problem.
  - Every write to a parent combines both children.
  - Writes containing more votes win.
- Max registers are easy in message-passing: use (Attiya, Bar-Noy, Dolev, 1995).
- Messages $= O(\text{size of quorum})$.

The full shared-coin algorithm

At each node, we track (count, variance, total) in a max register ordered by count.

Each process repeats:

1. Generate a new vote $v = \pm w$ (initally ± 1).
2. Add $(1, w^2, v)$ to local (count, variance, total).
3. For each ancestor I am scheduled to update this iteration:
   3.1 Read (count, variance, total) from both its children.
   3.2 Write sum of counts, variance, and total to ancestor's max register.
4. If I have done $4n \log_2 n$ votes since I last doubled my weight, set $w \leftarrow 2 \cdot w$.
5. If I just updated the root:
   5.1 Read (count, variance, total) from root.
   5.2 If variance $\geq n^2 \log_2 n$: return sgn(total).

Analysis: error due to missing votes

Basic idea is same as Bracha-Rachman: $|V^t_{\mathrm{root}} - U^t_{\mathrm{root}}|$ should be small.

- Why are they different?
- $U^t_x = U^{t0}_{x0} + U^{t1}_{x1}$ where $x0$ and $x1$ are children of x and $t0, t1$ are times in the past.
- So U^t_x is missing votes from between $t0$ and t and $t1$ and t.
- Similarly, U^{t0}_{x0} is missing votes from a similar gap between $t0$ and when $x0$'s children are read.
- Expanding this out recursively shows that all missing votes are accounted for by these missing intervals.

But there are only polynomially-many such intervals, so w.h.p. *every* interval with variance v has sum $O(\sqrt{v \log n})$.

After some inequality-crunching, *total* error is $O(n\sqrt{\log n})$.

Analysis: variance and costs

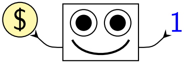
Total messages:

- With error $O(n\sqrt{\log n})$, we need $O((n\sqrt{\log n})^2) = O(n^2 \log n)$ variance.
- This translates into $O(n^2 \log n)$ total votes.
- Each vote has $O(\log n)$ amortized message overhead.
- $O(n^2 \log^2 n)$ messages total.

Individual messages:

- If I have to generate $O(n^2 \log n)$ variance by myself, I may have to double my votes $\Theta(\log n)$ times.
- Each doubling happens after $O(n \log n)$ votes.
- So I alone may generate $O(n \log^2 n)$ votes.
- $O(n \log^3 n)$ messages for me.

Conclusion



- We have shown how to implement randomized consensus in asynchronous message-passing with
  - $O(n^2 \log^2 n)$ messages total.
  - $O(n \log^3 n)$ messages per process.
- Corresponding lower bounds are $\Omega(n^2)$ and $\Omega(n)$.
- Can we get rid of the extra log factors?
- Can we use selective propagation idea for other problems?