Randomized Fault-Detecting Leader Election in a Bi-Directional Ring

Neal R. Wagner*
The University of Texas at San Antonio
Division of Mathematics, Computer Science, and Statistics
San Antonio, Texas 78285

Abstract. This article presents a randomized algorithm for leader election which uses bi-directionality of an asynchronous ring to force a node to "commit" to a coin flip by sending it in both directions. The algorithm is fault-detecting in a strong sense: it works if and only if there is a connected segment of $\lceil n/2 \rceil + 1$ non-faulty processors (n = ring-size). Faulty processors may do anything to disrupt the algorithm—even communicate outside the ring and cooperate. The algorithm guarantees that each non-faulty processor in the segment either has probability $1/n$ of being elected leader or will receive a fault message.

1. Introduction and Motivation.

Standard algorithms for electing a leader in a ring fail if one or more of the processors is faulty or "cheats," i.e., tries to alter election probabilities or tries to disrupt the algorithm [Abra 89], [Itai 81]. (The interesting article [Gold 87] considers only lost message faults, while [Brem 89] considers general synchronous networks of bounded degree.) This article studies a coin-flipping protocol which detects all faults:

- Each processor flips an unbiased coin.
- Only the total **`H-T`** configuration on the ring determines the outcome. Faulty processors cannot use deterministic flips to change the probabilities unless they have prior knowledge of all other flips.
- The protocols force each processor to "commit" to a **`H`** or **`T`** flip before knowing the total configuration of flips on the ring.

One problem is how to get this "commitment" to a flip. For example, in an asynchronous uni-directional ring, a faulty processor could accumulate everyone else's flip before choosing its own. We can see three solutions:

(a) Use a bi-directional ring. After an initial **`Send`** in each direction, each processor blocks until a **`Receive`** in both directions. Then the processor forwards both flips.

(b) Use a bi-directional ring with an agreed upon clockwise direction. Each processor does an initial **`Send`** in the clockwise direction. Then it waits until a **`Receive`** from the clockwise direction before a counterclockwise **`Send`**.

(c) Use a synchronous uni-directional ring. Now every processor must commit to **`Send`** a flip, and cannot wait to accumulate flips.

Consider the following protocol:

> **Informal Coin-flip Algorithm.**
> (1) Each of n processors flips an unbiased coin and sends its result around the ring.
> (2) **if** the number of heads is odd **then** select **`H`**
> (or select the subset of those who flipped **`H`**)
> **else** select **`T`**
> (or select the subset of those who flipped **`T`**).

*This material is based in part upon work supported by the Texas Advanced Research Program.

This does an unbiased **H−T** coin flip, and it also selects a subset of the n processors: those who flipped **H** or those who flipped **T**. Repeatedly choosing subsets of "active" processors will elect a leader in $O(\log n)$ rounds, with bit complexity $O(n^2 \log n)$. Thus the bit complexity is much greater than [Abra 89] or [Gold 87], but the number of rounds and the elapsed time is comparable.

Now suppose a processor wants to cheat. In approach (a) above, the processor can wait for coin flips from both sides and then choose its flip deterministically. It turns out that its best strategy is to choose the opposite of the incoming flips if they are the same. By this method a processor can improve its chances of election slightly above $1/n$.

The algorithms below yield probability of leader election exactly equal to $1/n$, even in the presence of this kind of cheating, by choosing the subset of those who flipped **H** with probability equal to the proportion of processors flipping **H**. This is accomplished by a sequence of coin flips used to approximate a random real number, agreed upon by all processors in the ring.

Similar remarks apply to approach (b). Approach (c) also solves these problems, but it has the disadvantage of requiring synchronous protocols and will not be discussed further here.

The algorithms below tolerate up to $\lfloor n/2 \rfloor - 1$ many faulty processors in a connected segment, and require at least $\lceil n/2 \rceil + 1$ non-faulty ones in a connected segment. These bounds are tight. We assume the faulty processors have arbitrary computing power and that they might cooperate by exchanging information over a channel separate from the ring. Thus even two cooperating faulty processors could isolate two segments of non-faulty processors of length less than $\lceil n/2 \rceil + 1$ and simulate the behavior of the rest of the ring to these segments. (So a non-faulty processor outside the main segment of length at least $\lceil n/2 \rceil + 1$ has no guarantees at all.) We assume that faulty processors do *not* have access to a ring segment of non-faulty processors except at the ends.

2. Algorithm Descriptions.

Configuration. Suppose there are n processors P_i, $0 \leq i \leq n-1$, arranged in a bi-directional asynchronous ring, with P_i connected to P_{i-1} and to P_{i+1}, using arithmetic $\bmod n$. Assume there are as least $\lceil n/2 \rceil + 1$ non-faulty processors in a connected segment, and assume the ring size n is known to each processor. First consider the following basis for our other algorithms (sample code for this algorithm appears in an Appendix):

Basic Algorithm (B). Each processor flips a coin, and all agree on each others' results.

(1) Each P_i independently chooses an unbiased flip, **H** or **T** (0 or 1).

(2) Each P_i sends the flip to P_{i-1} and to P_{i+1} (1 bit message; 2 bits including a possible fault message).

(3) Each processor blocks until the flips are received in both directions, and then forwards the coin flips.

(4) Repeat steps (2) and (3) until each P_i's flip goes completely around the ring in both directions.

(5) Each P_i keeps a record of the flip values from both directions, in 2 arrays of n bits, along with a record of the original flip.

(6) If the arrays do not agree, or the last value received in either direction is not the one sent out, then **failure** due to faulty processor. Initiate and send a fault message in each direction.

(7) In case there are no faults, denote the number of heads by n_H, the number of tails by n_T. Then one has $n_H + n_T = n, 0 \leq n_H, n_T \leq n$.

(8) If a fault message comes in either direction, then forward once in the same direction. Do not send a fault message to a processor which has already sent a fault message.

The coin flip algorithm below relies only on the number of heads or tails, while the subset algorithm produces a subset of the set of active processors.

Coin Flip Algorithm (F). All processors agree on a single coin flip. Requires $n \geq 4$ ($n \geq 3$ if there is an agreed clockwise direction in the ring).

(1) Perform the Basic Algorithm (B).

(2) If n_H is odd then agree on **H** else agree on **T**.

Subset Algorithm (S). All processors agree on a subset of the m "active" processors. Inactive processors continue to participate in the Basic Algorithm (B) and the Coin Flip Algorithm (F). Good for any $m \geq 2$. Input to the algorithm is an array of n bits, with m of them set to 1, showing which processors are active.

(1) Perform the Basic Algorithm (B), and all agree on which of m_H active processors got **H** and which of m_T active processors got **T**, $m = m_H + m_T$, and $0 \leq m_H, m_T \leq m$. (The inactive processors produce flips, even though the value of the flip is not used. Use the input bit array to filter out the inactive processors.)

(2) If $m_H = m$ or $m_T = m$ then let the subset be all m processors, and terminate the algorithm.

(3) Otherwise set $r = m_H/m, 0 < r < 1$.

(4) Invoke the Coin Flip Algorithm (F) repeatedly (using all processors, not just the "active" ones) to produce bits of a random real number t until it is clear whether $t < r$ or $r < t$. (Produce bits $0.b_1b_2b_3 \ldots b_j$ until either $0.b_1b_2 \ldots b_j > r$ or $0.b_1b_2 \ldots b_j111... < r$.)

(5) If $t < r$ then let the subset be those who chose **H** else if $t > r$ then let the subset be those who chose **T**.

Leader Election Algorithm (L). A ring of processors elect a leader. Good for $n \geq 4$.

(1) Initially let all processors be active.

(2) Invoke the Subset Algorithm (S) repeatedly to produce successive subsets of the set of active processors until there is only one active processor (or only two adjacent active processors).

(3) To prevent a possible stalemate in the case of two adjacent processors, the Subset Algorithm must be modified so that these processors do not send their initial flips to each other, but only send them around the ring.

3. Types of Faults.

A single faulty processor might do something incorrect at any step of any algorithm or might halt and refuse to continue. An exhaustive list of faults that can occur follows:

Type DF, *Deterministic Flip:* At the start of any round, wait until flips from both directions are received and then choose **H** or **T** deterministically. (In the case of a ring with an agreed upon clockwise direction, wait until a flip from the clockwise direction is received and then choose **H** or **T** deterministically.)

Type IM, *Initiate Inconsistent Message:* At the start of any round, send **H** one way and **T** the other way.

Type CM, *Change Message:* At any point in the protocol, forward **T** when **H** is received or eventually received, and vice-versa. This includes the possibility of sending **H** (determined randomly or deterministically) before receipt of **T** (and vice-versa).

Type BR, *Broken Ring:* In either direction (or both directions), fail to initiate a message or fail to forward a message. Messages in the other direction are then also blocked at this processor, as required by the algorithm.

Type FM, *Fault Message:* Initiate a fault message when no fault has occurred. For example, a processor might do this if it was not happy about the way the leader election was going.

Type NF, *No Fault message:* Fail to initiate a fault message when a fault has occurred.

Type SP, *Simulate Processor:* Several processors cooperate to simulate the existence of a whole ring section.

Type SM, *StaleMate:* In case only two adjacent processors remain active, use Type DF faults to force an endless sequence of the selection of both processors as the new set of active processors.

We may have up to a connected segment of $\lfloor n/2 \rfloor - 1$ cooperating faulty processors. Postanalysis outside the ring will often trace a fault to one of two processors on the ring, since many faults would show up as a disagreement about what was sent and what was received. (A communications link fault that alters a message would look the same.)

If a faulty processor initiates and sends a fault message in one direction when there has been no fault (Type FM), this might result in no information about who initiated the fault message. The best one could do is to keep time stamps of fault message receipt for postanalysis.

In case of *non*-cooperating faulty processors, almost any fault (except Type DF) would be detected.

4. Results.

THEOREM 1. *Suppose the Coin Flip Algorithm (F) is carried out with at most a connected segment of* $\lfloor n/2 \rfloor - 1$ *cooperating faulty processors.*

(a) *In the presence of Type DF faults, the algorithm produces an unbiased* **H** *or* **T** *for* $n \geq 4$ *(for* $n \geq 3$ *if there is an agreed upon clockwise direction).*

(b) *In the presence of faults of Types IM, CM, FM, or NF there is a guarantee that every non-faulty processor on the main segment will receive a fault message.*

(c) *Type SP faults are eliminated since each processor knows the ring size, while Type BR faults require that there be some maximum allowed response time.*

THEOREM 2. *Suppose* n *is known to all processors. Suppose there is at most a connected segment of* $\lfloor n/2 \rfloor - 1$ *cooperating faulty processors. Perform the Leader Election Algorithm (L) to attempt to elect a leader.*

(a) *In the presence of Type DF, consistent SP, or SM faults, the algorithm will successfully elect a leader. Each non-faulty processor has probability* $1/n$ *of being elected leader, and the faulty processors cannot alter any of these probabilities.*

(b) *In a ring with a maximum allowed response time in which* n *is known to all processors, it is guaranteed that either an unbiased leader election occurs or a fault message is received by the whole connected segment of non-faulty processors.*

THEOREM 3. *The Leader Election Algorithm (L) concludes in* $O(\log n)$ *rounds on the average. The bit complexity is* $O(n^2 \log n)$ *on the average.*

In fact, simulations show that the number of subsets taken is easily bounded above by $\lceil \log_2 n \rceil + 2$, and each subset involves at most 3 coinflip rounds.

5. Outlines of Proofs.

It is easy to see that in the absence of faults the Coin Flip Algorithm (F) produces an unbiased flip. Suppose one has a segment of up to $\lfloor n/2 \rfloor - 1$ cooperating faulty processors. (Just the two faulty processors at the segment ends could simulate the whole segment.) In this case, before any of the faulty processors see the innermost flip on the non-faulty segment, they must all agree on a consistent collection of flips, or face an eventual fault message. Thus at least this innermost flip is independent of all other flips. Any deterministic flips of the faulty processors will be combined later with this unbiased flip, to produce an overall unbiased **H−T** outcome.

Let $f(m)$ denote the probability that a given node in a subset of size m will eventually be elected leader $(1 \leq m \leq n)$. We wish to show that $f(m) = 1/m$. Suppose at the next stage we choose subsets of sizes k and $m-k$ $(1 \leq k \leq m)$. Suppose the probability that our given node is in the subset of size k is p, $0 < p < 1$, say, and $1-p$ that it is in the subset of size $m-k$. If it is in the subset of size k, then its probability of leader election is $\frac{k}{m} \cdot f(k)$, where the $\frac{k}{m}$ factor comes from steps (3)–(5) of the Subset Algorithm. Then

$$\begin{aligned} f(m) &= p \cdot \frac{k}{m} \cdot f(k) \; + \; (1-p) \cdot \frac{m-k}{m} \cdot f(m-k) \\ &= p \cdot \frac{k}{m} \cdot \frac{1}{k} \; + \; (1-p) \cdot \frac{m-k}{m} \cdot \frac{1}{m-k} \\ &= \frac{1}{m}, \end{aligned}$$

by complete induction. It is interesting that the algorithm works no matter how the successive subsets are determined, as long as one does not keep selecting the whole active set.

The Subset Algorithm (S) will terminate in slightly less than three coin flips on the average since step (4) of it will involve one flip half the time, two flips a quarter of the time, three flips an eighth of the time, and so forth. Thus the *expected number of flips* in step (4) will be

$$\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \ldots + \frac{n}{2^n} + \ldots = 2.$$

Step (4) ensures that the two complementary subsets each has probability of selection equal to their relative sizes.

The Leader Election Algorithm (L) will halve the size of the active subset on the average at each stage, so it will require $O(\log n)$ coin flips or rounds.

6. Conclusions.

We have presented a simple leader election algorithm in a bi-directional asynchronous ring. The algorithm would be practical to implement and use. It is completely fault-detecting in the sense that a connected segment of up to $\lfloor n/2 \rfloor - 1$ faulty cooperating processors trying to alter or disrupt the algorithm have only two choices: (1) allow each of the non-faulty processors on the complementary segment probability $1/n$ of being elected leader (n = ringsize), or (2) face the fact that each non-faulty processor on the segment will receive a fault message. The bit complexity is much greater than methods which do not detect faults, but the elapsed time (= number of rounds) is of the same order.

Acknowledgment. The author wishes to thank Kay and Steve Robbins for encouragement.

References.

[Abra 89] K. Abrahamson, et al., "The bit complexity of randomized leader election in a ring," *Siam J. Computing* 18 (1, Feb. 1989), pp. 12–29.

[Berm 89] P. Berman and J. A. Garay, "Efficient agreement on bounded-degree networks," *1989 International Conference on Parallel Processing*, pp. I-188–I-191.

[Gold 87] O. Goldreich and L. Shrira, "Electing a leader in a ring with link failures," *Acta Informatica* 1987, pp. 79–91.

[Itai 81] A. Itai and M. Rodeh, "Symmetry breaking in distributed networks," *Proceedings of the 22nd Annual IEEE Symposium on Foundations of Computer Science*, 1981, pp. 150–158.

Appendix.

Sample code for the Basic Algorithm (B).

```
const n = 25; (* the ring size *)
type Message = (Heads, Tails, Fault);
   Bits = array[0..n − 1] of Heads..Tails;
function B (var R, L: Bits): boolean;
var PT : 0..n − 1;
   Flip: Heads..Tails;
   Faulty: Boolean;
begin
   Flip ← TrueRandomFlip();
   R[0] ← Flip; L[0] ← Flip;
   for PT ← 0 to n − 1 do
      parallelbegin (* if done sequentially,
                      must do sends first
                      to avoid deadlock *)
         SendRight(R[(n − PT) mod n]);
         SendLeft(L[PT]);
         ReceiveFromLeft(R[n − PT − 1]);
         ReceiveFromRight(L[(PT + 1) mod n])
      parallelend; (* last receives overwrite
                      0th table entries *)
   Faulty ← (Flip ≠ R[0]) or (Flip ≠ L[0]);
   for PT ← 0 to n − 1 do
      if R[PT] ≠ L[PT] then Faulty ← true;
   B ← Faulty;
end; (* B *)
(* Note: In case of a fault, send a fault
      message in each direction. *)
```