Correctness Proof of Ben-Or's Randomized Consensus Algorithm*

Marcos Kawazoe Aguilera Sam Toueg

May 17, 1998

Abstract

We present a correctness proof for Ben-Or's Randomized Consensus Algorithm for the case in which processes can fail by crashing, and a majority of processes is correct. This is the first time that the proof of Ben-Or's algorithm appears for this case. The proof has been extracted from [AT96]: it is a simplification of the correctness proof of a more complex consensus algorithm that involves both randomization and failure detection.

1 Introduction

We present a correctness proof for Ben-Or's Randomized Consensus Algorithm for the case in which processes can fail by crashing, and $n > 2f$ (n is the number of processes and f is the maximum number of processes that can crash). This is the first time that the proof of Ben-Or's algorithm appears for this case. Previous proofs either assume that $n > 3f$ [Lyn96] or that the adversary is weak [Had91].

Our proof has been extracted from [AT96]. That paper gives a hybrid consensus algorithm that uses both randomization and failure detection. The proof that we present here is a simplification of the proof in [AT96]: roughly speaking, we obtained it by "removing" the failure detection part of the proof in that paper.

2 Informal Model

Our model of asynchronous computation is patterned after the one in [FLP85]. We only sketch its main features here. We consider *asynchronous* distributed systems in which there is no bound on message delay, clock drift, or the time necessary to execute a step. To simplify the presentation of our model, we assume the existence of a discrete global clock. This is merely a fictional device: the processes do not have access to it. We take the range $\mathcal{T}$ of the clock's ticks to be the set of natural numbers $\mathbb{N}$.

The system consists of a set of n *processes*, $\Pi = \{p_0, p_1, \ldots, p_{n-1}\}$. Every pair of processes is connected by a reliable communication channel. Up to f processes can fail by *crashing*.

Each process has access to a random number generator (r.n.g.). When a process queries its r.n.g. it obtains a random bit. For simplicity, we assume a uniform distribution, i.e., the r.n.g. outputs 0 and 1, each with probability 1/2.

*Research partially supported by NSF grant CCR-9402896, by ARPA/ONR grant N00014-96-1-1014, and by an Olin Fellowship.

A distributed algorithm $\mathcal{A}$ is a collection of n deterministic automata (one for each process in the system) that communicate by sending messages through reliable channels. The execution of $\mathcal{A}$ occurs in *steps* as follows. For every time $t \in \mathcal{T}$, at most one process takes a step. Each step consists of receiving a message; querying the r.n.g.; changing state; and optionally sending a message to one process. We assume that messages are never lost. That is, if a process does not crash, it eventually receives every message sent to it.

A schedule is a sequence $\{s_j\}_{j\in\mathbb{N}}$ of processes and a sequence $\{t_j\}_{j\in\mathbb{N}}$ of strictly increasing times. A schedule indicates which processes take a step and when: for each j, process s_j takes a step at time t_j. A process *crashes* in a schedule if it takes only a finite number of steps. If a process does not crash, we say that is it *correct*.

2.1 Adversary Power

When designing fault-tolerant algorithms, we often assume that an adversary has some control on the behavior of the system, e.g., the adversary may be able to control the occurrence and the timing of process failures, the message delays, and the scheduling of processes. Adversaries may have limitations on their computing power and on the information that they can obtain from the system. Different algorithms are designed to defeat different types of adversaries [CD89].

We now describe the adversary that Ben-Or's algorithm defeats. The adversary has unbounded computational power, and full knowledge of all process steps that already occurred. In particular, it knows the contents of all past messages, the internal state of all processes in the system,[1] and all the previous outputs of the r.n.g.. With this information, at any time in the execution, the adversary can dynamically select which process takes the next step and which message this process receives (if any). The adversary, however, operates under the following restrictions: every message sent to a correct process must eventually be received and the final schedule may have at most f crashed processes.

3 The Consensus Problem

In the *uniform binary consensus* problem every process p has some *initial value* $v_p \in \{0, 1\}$, and must *decide* on a value such that:

Uniform agreement: If processes p and p' decide v and v', respectively, then $v = v'$;

Uniform validity: If some process decides v, then v is the initial value of some process;

Termination: Every correct process eventually decides some value.

For probabilistic consensus algorithms, Termination is weakened to

Termination with probability 1: With probability 1, every correct process eventually decides some value.

[1] This is in contrast to the assumptions made by several algorithms, e.g., those that use cryptographic techniques.

Every process p executes the following:

procedure *consensus*(v_p) $\{v_p$ is the initial value of process $p\}$

  $x \leftarrow v_p$ $\{x$ is p's current estimate of the decision value$\}$
  $k \leftarrow 0$

  while true **do**
    $k \leftarrow k + 1$ $\{k$ is the current phase number$\}$
    send (R, k, x) **to** all processes

    wait for messages of the form $(R, k, *)$ **from** $n - f$ processes $\{$"$*$" can be 0 or 1$\}$
    if received more than $n/2$ (R, k, v) with the same v
    then send (P, k, v) **to** all processes
    else send $(P, k, ?)$ **to** all processes

    wait for messages of the form $(P, k, *)$ from $n - f$ processes $\{$"$*$" can be 0, 1 or ?$\}$
    if received at least $f + 1$ (P, k, v) with the same $v \neq ?$ **then** *decide*(v)
    if at least one (P, k, v) with $v \neq ?$ **then** $x \leftarrow v$ **else** $x \leftarrow 0$ or 1 randomly {query r.n.g.}

Figure 1: Ben-Or's Randomized Consensus algorithm

4 Ben-Or's Randomized Consensus Algorithm

The Randomized Consensus algorithm shown in Figure 1 is due to Ben-Or [Ben83]. The algorithm works under the assumption that a majority of processes are correct (i.e., $n > 2f$). It is easy to see that this requirement is necessary for any algorithm that solves Consensus in asynchronous systems with crash failures, even if all processes have access to a random number generator.

In the algorithm, every message contains a tag (R or P), a phase number, and a value which is either 0 or 1 (for messages tagged P, it could also be "?"). Messages tagged R are called *reports* and those tagged with P are called *proposals*. When p sends (R, k, v) or (P, k, v) we say that p *reports* or *proposes* v in phase k, respectively.

Each execution of the **while** loop is called a *phase*, and each phase consists of two asynchronous rounds. In the first round, processes report to each other their current estimate (0 or 1) for a decision value.

In the second round, if a process receives a majority of reports for the *same* value then it proposes that value to all processes, otherwise it proposes "?". Note that it is impossible for one process to propose 0 and another process to propose 1 in the same phase. At the end of the second round, if a process receives $f + 1$ proposals for the same value different than ?, then it decides that value. If it receives at least one value different than ?, then it adopts that value as its new estimate, otherwise it adopts a random value for its estimate.

The algorithm in Figure 1 does not include a halt statement. Moreover, once a correct process decides a value, it will keep deciding the same value in all subsequent phases. However, it is easy to modify the algorithm so that every process decides at most once, and halts at most one round after deciding.

5 Proof of Correctness

Assume that there is a majority of correct processes (i.e., $n > 2f$). We show the following:

Theorem 1

(Safety) Ben-Or's algorithm always satisfies uniform validity and uniform agreement.

(Liveness) The algorithm satisfies termination with probability 1.

Proof: We say that *process p starts phase k* if process p completes at least $k-1$ iterations of the **while** loop. We say that *process p reaches line n in phase k* if process p starts phase k and p executes past line $n-1$ in that phase. We say that *v is k-locked* if every process that starts phase k does so with its variable x set to v. When ambiguities may arise, a local variable of a process p is subscripted by p, e.g., x_p is the local variable x of process p.

We first show the safety properties.

Lemma 1 *Suppose $k > 0$. Then it is impossible for a process to propose 0 and another one to propose 1 in the same phase k.*

Proof: We prove the result by contradiction: suppose that two processes p and q propose 0 and 1, respectively, in phase k. Thus, p received more than $n/2$ reports for 0 and q received more than $n/2$ reports for 1 in phase k. But then there is a process that reports 0 to p and 1 to q in phase k, and this is impossible. □

Lemma 2 *If some process decides v in phase $k > 0$, then v is $(k+1)$-locked.*

Proof: Suppose some process p decides v in phase $k > 0$ (note that $v \neq ?$), and let q be any process that starts phase $k+1$. From the algorithm, p receives at least $f+1$ proposals for v in phase k (line 10). In phase k, q waits for $n-f$ proposals in line 10. Because p receives $f+1$ proposals for v, q must have received at least one proposal for v. Moreover, by Lemma 1, q does not receive any proposals for $\bar{v}$.[2] So q sets x_q to v in line 12 and starts phase $k+1$ with $x_q = v$. □

Lemma 3 *If a value v is k-locked for some $k > 0$, then every process that reaches line 12 in phase k decides v in phase k.*

Proof: Suppose v is k-locked for some $k > 0$. Then, all reports sent in line 6 of phase k are for v. Since $n-f > n/2$, every process that proposes some value in phase k proposes v in line 8. Consider a process p that reaches line 12 in phase k. Clearly, p receives $n-f$ proposals (line 10) for v in phase k. Since $n-f \geq f+1$, p decides v in phase k. □

Corollary 1 *If some process decides v in phase $k > 0$, then every process that reaches line 12 in phase $k+1$ decides v in phase $k+1$.*

Proof: By Lemma 2 and Lemma 3. □

[2] We denote by $\bar{v}$ the binary complement of bit v.

function FavorableToss(r, u): **bit** $\{$evaluated only at time $u \geq \tau_k$ where $k = 2r\}$

$k \leftarrow 2r$ $\{k$ is the first phase in epoch $r\}$

if some value v is k-major at time τ_k **then return** v

if by time u no process received $n - f$ proposals in phase $k + 1$ **then return** 0 $\{u < \tau_{k+1}\}$

if at time τ_{k+1} 0 is $(k + 1)$-major **then return** 0 $\{$here $u \geq \tau_{k+1}\}$
else return 1

Figure 2: Favorable coin toss algorithm

Corollary 2 *(Uniform agreement) If some processes p and p' decide v and v' in phase $k > 0$ and $k' > 0$, respectively, then $v = v'$.*

Proof: For $k = k'$ the result follows from Lemma 1 and the fact that a process can decide a value in a phase only if that value was proposed in the same phase. Assume that $k < k'$. Since p' decides in phase k' then p' reaches line 12 in every phase r, $k < r \leq k'$. Since p decides v in phase k, by Corollary 1 p' decides v in phase $k + 1 \leq k'$. By additional applications of Corollary 1, we conclude that p' decides v in phase k'. Each process can decide at most once per phase, so $v = v'$. □

Corollary 3 *(Uniform validity) If some process p decides v, then v is the initial value of some process.*

Proof: Note that $v \in \{0, 1\}$. If the initial values of all processes are not identical, then v is clearly the initial value of some process. Now, suppose all processes have the same initial value w. Thus, w is 1-locked. From Lemma 3, p decides w, and from Corollary 2, $w = v$. □

We now proceed to show that the algorithm satisfies termination with probability 1.

Lemma 4 *Every correct process starts every phase $k > 0$.*

Proof: The detailed proof is by a simple but tedious induction on k. We describe only the central idea here. In each phase, there are two **wait** statements that can potentially block processes (lines 6 and 10). However, it is not possible for a correct process to be blocked in any of these statements because at least $n - f$ processes are correct and send the messages that this process is waiting for. □

Corollary 4 *If a value v is k-locked for some $k > 0$, then every correct process decides v in phase k.*

Proof: Immediate from Lemmata 3 and 4. □

Corollary 5 *If some process decides v in phase $k > 0$, then every correct process decides v in phase $k + 1$ (and thus in all subsequent phases).*

Proof: Immediate from Corollary 1 and Lemma 4. □

For $k > 0$, let τ_k be the first time that any process receives $n - f$ proposals in phase k. From Lemma 4, for every $k > 0$, some process receives $n-f$ proposals in phase k, and so τ_k is well-defined. Note that in the algorithm no process queries the r.n.g. in phase k before time τ_k.

For each $k > 0$, we say that a value v *is k-major at time t* if by time t more than $n/2$ processes have started phase k with their variable x set to v.[3] Clearly, for each $k > 0$ and all times t and t', it is impossible for 0 to be k-major at t, and 1 to be k-major at t'.

Consider a process p that sets x_p to v in line 12 of phase k. If v was obtained from the r.n.g., we say that p *R-gets v in phase k*; otherwise, we say that p *D-gets v in phase k*.

Lemma 5 *For every $k \geq 1$: (1) if some process D-gets v in phase k, then v is k-major at some time; (2) if v is ever k-major, then v is the only value that a process can D-get in phase k.*

Proof: Consider phase $k \geq 1$. Suppose p D-gets v. Then p received at least one proposal for v from some process q. So more than $n/2$ processes must have reported v to q in phase k. Thus, v was k-major — proving part (1). Part (2) follows from part (1) and the fact that v and $\bar{v}$ cannot both be k-major. □

For the rest of the proof, we group pairs of phases into *epochs* as follows: *epoch r* consists of phases $2r$ and $2r + 1$.[4] We will define the concept of a "lucky" epoch — one in which processes toss coins that cause the termination of the algorithm (no matter what the adversary does). To do so, we first define function *FavorableToss*(r, u) given in Figure 2. We say that *epoch r is lucky* if, for every process p and any time u, if p queries the r.n.g. in epoch r at time u, then p obtains *FavorableToss*(r, u) from the r.n.g.. Note that if p queries the r.n.g. in epoch r at time u, this occurs after at least one process receives $n - f$ proposals in phase $2r$. Thus, $\tau_{2r} \leq u$. So it is clear that the code of *FavorableToss*(r, u) refers only to what occurred in the system up to time u.

Lemma 6 *For every $r \geq 1$, if epoch r is lucky then some value is $(2r+1)$-locked or $(2r+2)$-locked.*

Proof: Throughout the proof of this lemma, fix some arbitrary $r \geq 1$ and assume that epoch r is lucky. Let $k = 2r$; recall that epoch r consists of phases k and $k + 1$. Since epoch r is lucky, if any process R-gets a value v at time t and in phase $j = k$ or $j = k + 1$, then $v =$ *FavorableToss*(r, t) and $\tau_k \leq t$.

Case 1: Suppose some value v is k-major at time τ_k. By the definition of *FavorableToss*, for any u such that $\tau_k \leq u$, *FavorableToss*$(r, u) = v$. So, v is the only value that a process can R-get in phase k. Since v is k-major, by Lemma 5, v is the only value that a process can D-get in phase k. Thus, v is $(k + 1)$-locked.

Case 2: Now assume that no value is k-major at time τ_k.

Case 2.1: Suppose that 0 is $(k + 1)$-major at time τ_{k+1}. By the definition of *FavorableToss*, for any u such that $\tau_{k+1} \leq u$, *FavorableToss*$(r, u) = 0$. So, 0 is the only value that a process can R-get in phase $k+1$. Since 0 is $(k+1)$-major, by Lemma 5, 0 is the only value that a process can D-get in phase $k + 1$. Thus, 0 is $(k + 2)$-locked.

Case 2.2: Now assume that 0 is not $(k + 1)$-major at time τ_{k+1}. By time τ_{k+1}, a majority of processes started round $k + 1$. Since 0 is not $(k + 1)$-major at time τ_{k+1}, by this time some process p starts round $k + 1$ with x_p set to 1; thus either p D-got 1 or R-got 1 in phase k. However, for $\tau_k \leq u < \tau_{k+1}$ we have *FavorableToss*$(r, u) = 0$, and so p D-got 1 in phase k. By Lemma 5, 1 was k-major.

[3] Recall that a process starts phase k when it completes $k - 1$ iterations of the loop, i.e., right after it executes line 12.

[4] Phase 1 is not part of any epoch.

Since 1 was k-major, by Lemma 5, 1 is the only value that a process can D-get in phase k. Moreover, for $\tau_{k+1} \leq u$, we have $FavorableToss(r, u) = 1$, and so 1 is the only value that a process can R-get in phase k at or after time τ_{k+1},. Thus, after time τ_{k+1}, 1 is the only value that a process can D-get or R-get in phase k. So, after time τ_{k+1}, no process starts round $k+1$ with x_p set to 0. Since at time τ_{k+1}, 0 is not $(k+1)$-major, we conclude that 0 is never $(k+1)$-major.

Since 0 is never $(k+1)$-major, by Lemma 5, 1 is the only value that a process can D-get in phase $k+1$. Moreover, for $\tau_{k+1} \leq u$, we have $FavorableToss(r, u) = 1$, and so 1 is the only value that a process can R-get in phase $k+1$. Thus, 1 is $(k+2)$-locked. $\square$

Lemma 7 *The probability that some epoch is lucky is 1.*

Proof: The result is immediate from the following observation: for every $r \geq 1$, (a) the probability that epoch r is lucky is at least 2^{-2n} (because in each phase there are at most n queries to the r.n.g.), and (b) for any $r' \neq r$, the events "epoch r is lucky" and "epoch r' is lucky" are independent (because epochs r and r' consist of disjoint sets of phases).$\square$

Lemma 8 *(Termination with probability 1) The probability that all correct processes decide is 1.*

Proof: Immediate from Lemmata 7 and 6, and Corollary 4.$\square$

The proof of Theorem 1 is now complete: uniform validity and uniform agreement were shown in Corollary 3 and Corollary 2, respectively. Termination with probability 1 was shown in Lemma 8.$\square_{Theorem\ 1}$

From the proof of Lemma 7, it is easy to see that the expected number of rounds for termination is $O(2^{2n})$.

References

[AT96] Marcos Kawazoe Aguilera and Sam Toueg. Randomization and failure detection: a hybrid approach to solve consensus. In *Proceedings of the 10th International Workshop on Distributed Algorithms*, Lecture Notes on Computer Science, pages 29–39. Springer-Verlag, October 1996.

[Ben83] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the Second ACM Symposium on Principles of Distributed Computing*, pages 27–30, August 1983.

[CD89] Benny Chor and Cynthia Dwork. Randomization in Byzantine Agreement. *Advances in Computer Research (JAI Press Inc.)*, 4:443–497, 1989.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[Had91] Vassos Hadzilacos. Lecture notes. Unpublished manuscript., 1991.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., 1996.