Distrib. Comput.
DOI 10.1007/s00446-012-0162-z

The correctness proof of Ben-Or's randomized consensus algorithm

Marcos K. Aguilera · Sam Toueg

Received: 13 October 2010 / Accepted: 17 November 2011

Abstract In a ground-breaking paper that appeared in 1983, Ben-Or presented the first randomized algorithm to solve consensus in an asynchronous message-passing system where processes can fail by crashing. Although more efficient randomized algorithms were subsequently proposed, Ben-Or's algorithm is still the simplest and most elegant one. For this reason, it is often taught in distributed computing courses and it appears in several textbooks. Even though Ben-Or's algorithm is widely known and it is very simple, surprisingly a proof of correctness of the algorithm has not yet appeared: previously published proofs make some simplifying assumptions—specifically, they either assume that $f < n/3$ (n is the total number of processes and f is maximum number of processes that may crash) or that the adversary is weak, that is, it cannot see the process states or the content of the messages. In this paper, we present a correctness proof for Ben-Or's randomized consensus algorithm for the case that $f < n/2$ process crashes and the adversary is strong (i.e., it can see the process states and message contents, and schedule the process steps and message receipts accordingly). To the best of our knowledge, this is the first full proof of this classical algorithm. We also demonstrate a counterintuitive problem that may occur if one uses the well-known abstraction of a "global coin" to modularize and speed up randomized consensus algorithms, such as Ben-Or's algorithm. Specifically, we show that contrary to common belief, the use of a global coin can sometimes be deleterious rather than beneficial: instead of speeding up Ben-Or's algorithm, the use of a global coin in this algorithm may actually prevent termination.

M. K. Aguilera (✉)
Microsoft Research Silicon Valley, 1065 La Avenida,
Mountain View, CA 94043, USA
e-mail: marcos_aguilera_msrsvc@live.com

S. Toueg
University of Toronto, 10 King's College Road,
Toronto, ON M5S 3G4, Canada

1 Introduction

The *consensus* problem, i.e., the problem of processes agreeing on an input value, is at the core of many distributed problems in both theory and practice. In particular, solving consensus is central to the state machine approach used to build fault-tolerant services [15]. While consensus can be solved in synchronous systems for various types of failures (e.g., see [7]), it is well-known that this problem cannot be solved in asynchronous message-passing systems, even if we assume that (a) all communication links are reliable, (b) at most one process can fail, and (c) a process can fail only by crashing [11].

Soon after this fundamental impossibility result appeared, Ben-Or pioneered the use of randomization to circumvent this result. In particular, in a ground-breaking paper [8], Ben-Or gave the first randomized algorithm that solves consensus "with probability 1" in an asynchronous message-passing system with $f < n/2$, where n is the total number of processes and f is the maximum number of processes that may crash.

Even though Ben-Or's algorithm is widely known and it is very simple, to the best of our knowledge a full proof of correctness has not yet appeared: previously published proofs make some simplifying assumptions; specifically, they assume either that $f < n/3$ [14] or that the adversary is weak[1] [4,13]. The presence of such assumptions in current proofs leaves an undesirable gap in the knowledge of randomized algorithms and raises the following questions: does Ben Or's classical randomized consensus algorithm work without these assumptions? If it works, what are the key reasons for its correctness? If it does not work, why does it fail?

In this paper, we prove that Ben-Or's randomized consensus algorithm works without simplifying assumptions; more precisely, we present a correctness proof for the case that $f < n/2$ and the adversary is strong. Intuitively, a strong adversary is one that can see the process states and message contents, and schedule the process steps and message receipts accordingly. This is the first full proof of this classical algorithm, which is still the simplest randomized consensus algorithm for message-passing systems with crash failures.

We also demonstrate a counterintuitive problem that may occur if one uses the well-known abstraction of a "global coin" to modularize and speed up randomized consensus algorithms, such as Ben-Or's algorithm. In many randomized consensus algorithms, processes toss coins, and the hope is that many processes get the same outcome: if this occurs, consensus is easily reached and the algorithm terminates. So a common technique to speed up such algorithms is to replace the independent coins of the processes by a *global coin* with a parameter $\rho > 0$; this is a coin such that, for each possible outcome $v \in \{0, 1\}$, with probability at least ρ, all the processes that toss the coin get the same outcome v [7]. With this technique, an algorithm is first shown to be correct assuming that processes have access to a global coin with parameter ρ, and then this global coin is implemented and proven correct. Intuitively, the higher the parameter ρ of the global coin is, the faster agreement and termination occurs.

We show that this intuition can be flawed by exposing a subtle pitfall associated with the use of a global coin. To do so, we consider the global-coin variant of Ben-Or's algorithm: this is exactly like Ben-Or's algorithm except that we substitute the independent coins of the processes by a global coin with parameter $\rho = 1/2$, i.e., when processes toss this coin, with probability 1/2 they all get 0, and with probability 1/2 they all get 1. Intuitively, this substitution should reduce the expected termination time of the algorithm. But does it really do so?

We prove here that, *under a strong adversary*, the global-coin variant of Ben-Or's algorithm actually does *not* work: in fact, with probability at least 1/2, it does not terminate. We prove this by constructing a run where, if the strong adversary is lucky for the first coin toss in the first round (this occurs with probability 1/2), it can thereafter schedule the process steps and message receipts in a way that prevents processes from ever deciding. The construction of this run is somewhat subtle, and it illustrates a potential pitfall of using the abstraction of a global coin in conjunction with a strong adversary.

In summary, the contribution of this paper is twofold:

(1) We present a proof that Ben-Or's randomized consensus algorithm works against a strong adversary for any $f < n/2$ crash failures. To the best of our knowledge, this is the first such proof for this classical and simple algorithm.
(2) We prove that the *global-coin variant* of Ben-Or's algorithm does *not* work against a strong adversary for any f such that $n/3 \leq f < n/2$. This shows that, contrary to a common belief, replacing independent coins with a global coin (in a randomized consensus algorithm) can sometimes be deleterious rather than beneficial: instead of speeding up the algorithm, the use of a global coin may actually prevent termination.

Note that if $f < n/3$, then the global-coin variant of Ben-Or's algorithm *does* work against a strong adversary: the proof is very similar to the one of Ben-Or's algorithm given in [14, Chapter 21.3] for the case that $f < n/3$. Together with our result (2) above, this implies that the global-coin variant of Ben-Or's algorithm works against a strong adversary *if and only if* $f < n/3$.

As a final remark, we note that our proof that Ben-Or's algorithm works under a strong adversary for any $f < n/2$ is not as simple as one may expect given the simplicity of the algorithm itself. In fact, the arguments that we use in this proof encompass what happens over pairs of consecutive rounds (reasoning one round at a time is not sufficient). But the complexity of this proof is probably not accidental: any proof of correctness must work when processes use independent coins, but it *must* also fail when we substitute these coins with a global coin in the case that $n/3 \leq f < n/2$, and the reason for this failure is somewhat subtle (as illustrated by the non-terminating run that we construct in Sect. 7).

A preliminary version of this paper appeared as a technical report [6]. Since Ben-Or's message-passing randomized consensus algorithm was published, many other randomized consensus algorithms appeared, especially for shared-memory systems. In Sect. 8, we briefly discuss some recent work in this area.

Roadmap We first describe our model in Sect. 2 and the consensus problem in Sect. 3. We then present Ben-Or's randomized consensus algorithm in Sect. 4, and its proof of correctness, under a strong adversary and any $f < n/2$, in Sect. 5. In Sect. 6 we describe the global-coin variant of Ben-Or's algorithm, and in Sect. 7 we prove that it does *not* terminate under a strong adversary and $f \geq n/3$. In Sect. 8, we briefly discuss some recent work in the area of

[1] Intuitively, a weak adversary is a scheduler that cannot see the process states or the content of the messages.

randomized consensus. We conclude the paper with some remarks in Sect. 9.

2 Informal model

Our model is patterned after the one in [11] and we only sketch its main features here. We consider asynchronous message-passing distributed systems where processes may crash but communication channels are reliable. In these systems, there is no bound on message delay, clock drift, or the time a process takes to execute a step, but every message sent to a non-faulty process is eventually received. To simplify the presentation, 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 $\mathbb{N}$.

There are $n \geq 2$ processes, $\Pi = \{p_1, \ldots, p_n\}$, at most f of them may crash, and every pair of processes is connected by a reliable communication channel. Each process has access to a coin that it can toss to obtain independent random bits. For simplicity, we assume a uniform distribution, i.e., when a process tosses its coin, it obtains 0 or 1, each with probability $1/2$. In Sect. 6, we also consider *global coins*, which we define precisely in that section.

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 the 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; optionally tossing a coin; changing state; and optionally sending a message to one process. Since channels are reliable, messages are never lost: 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 (in that schedule). If a process does not crash, we say that is it correct.

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 (e.g., see [10]).

The results in this paper concern a *strong adversary*, which has unbounded computational power and full knowledge of all process steps that already occurred. In particular, it knows the contents of all messages sent, the internal state of all processes in the system, and the outcome of all the previous coin tosses. 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 *binary consensus* problem every process p has some *initial value* $v_p \in \{0, 1\}$, and must *decide* on a value such that the following safety and liveness properties hold.

(Safety properties)

– *Uniform agreement:* No two processes decide differently.
– *Uniform validity:* If any process decides v, then v is the initial value of some process.

(Liveness property)

– *Termination:* Every correct process eventually decides.

For randomized consensus algorithms, the liveness property is weakened to

– *Termination with probability 1:* With probability 1, every correct process eventually decides.

Henceforth, when we say "consensus", we refer to the binary consensus problem where the liveness property is termination with probability 1.

4 Ben-Or's randomized consensus algorithm

Figure 1 shows Ben-Or's randomized consensus algorithm for crash failures [8]. The algorithm assumes that a majority of processes are correct, i.e., it assumes that $f < n/2$. As Ben-Or remarked in [8], the assumption that $f < n/2$ is necessary for any algorithm that solves consensus in asynchronous systems with crash failures, even if processes can toss random coins. The proof of this is based on a standard partition argument [8].

Ben-Or's randomized consensus algorithm, shown in Fig. 1, is structured as a **while** loop. Each iteration of this loop is an (asynchronous) *round*. In each round, processes exchange messages twice, and each message exchange is called a *phase*. Every message contains a tag (R or P),

```
Every process $p$ executes the following:
0  procedure consensus($v_p$)                                  {$v_p$ is the initial value of process $p$}
1      $x \leftarrow v_p$                                      {$x$ is $p$'s current estimate of the decision value}
2      $k \leftarrow 0$
3      while true do
4          $k \leftarrow k+1$                                  {$k$ is the current round number}
5          send $(R, k, x)$ to all processes
6          wait for messages of the form $(R, k, *)$ from $n-f$ processes      {"$*$" can be 0 or 1}
7          if received more than $n/2$ $(R, k, v)$ with the same $v$
8          then send $(P, k, v)$ to all processes
9          else send $(P, k, ?)$ to all processes
10         wait for messages of the form $(P, k, *)$ from $n-f$ processes      {"$*$" can be 0, 1 or ?}
11         if received at least $f+1$ $(P, k, v)$ with the same $v \neq ?$ then decide $v$
12         if received at least one $(P, k, v)$ with $v \neq ?$ then $x \leftarrow v$ else $x \leftarrow$ 0 or 1 randomly      { toss coin }
```

Fig. 1 Ben-Or's randomized consensus algorithm

a round number, and a value which is either 0 or 1; for messages tagged P, it could also be "?". Messages tagged R are sent in the first phase of a round and are called *reports*. Messages tagged with P are sent in the second phase of a round and are called *proposals*. When p sends (R, k, v) or (P, k, v) we say that *p reports* or *proposes v in round k*, respectively.

In the first phase of each round, processes report to each other their current estimate (0 or 1) for a decision value. In the second phase, if a process receives a strict majority of reports for the *same* value v then it proposes v 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 round. At the end of the second phase, if a process receives $f+1$ proposals for the same value v different than ?, then it decides v. If it receives at least one value v different than ?, then it adopts v as its new estimate; otherwise it adopts a random bit for its estimate.

The algorithm in Fig. 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 rounds. However, it is easy to modify the algorithm so that every process decides at most once and halts.[2]

5 Correctness of Ben-Or's algorithm under a strong adversary and $f < n/2$

In this section, we present a proof that Ben-Or's algorithm solves consensus under a strong adversary in a system where $f < n/2$. In this proof, we first show that Ben-Or's algorithm satisfies the validity and agreement properties of consensus; this part of the proof is quite simple and standard. We then show that Ben-Or's algorithm satisfies termination with probability 1. This part of the proof is more complex, due to the power of the adversary and our assumption that almost half of the processes may fail.[3]

In the following, we say that *process p starts round k* if process p completes at least $k-1$ iterations of the **while** loop. We say that *process p reaches line j in round k* if process p starts round k and p executes past line $j-1$ in that round. We say that *v is k-locked* if every process that starts round 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. Throughout the proof, we assume that $f < n/2$, i.e., less than half of the processes may crash.

5.1 Agreement and validity

We first show that the algorithm in Fig. 1 satisfies the safety properties of consensus. In the proof, we use the fact that processes can report only 0 or 1, and propose only 0, 1, or ? in each round (this is easily shown by induction).

Lemma 1 *It is impossible for a process to propose* 0 *and a process to propose* 1 *in the same round* $k \geq 1$.

Proof The proof is by contradiction. Suppose that processes p and p' propose 0 and 1, respectively, in round k. Thus, p received more than $n/2$ reports for 0 and p' received more than $n/2$ reports for 1 in round k. So there is a process that reports 0 to p and 1 to p' in round k, and this is impossible. □

[2] One way to do so is as follows: in line 11, if a process is about to decide some value v, it first sends a special message $(decide, v)$ to all processes, then it decides v and halts; any process that receives such a message sends it to all processes, decides v, and halts.

[3] Proofs that assume a weaker adversary [13] or fewer failures [14] are much simpler.

Lemma 2 *If some process decides v in round $k \geq 1$, then v is $(k+1)$-locked.*

Proof Suppose that some process p decides v in round $k \geq 1$ (note that this occurred in line 11 and $v \neq ?$). In round k, p must have received at least $f+1$ proposals for v in line 10. Let q be any process that starts round $k+1$. In round k, q received $n-f$ proposals in line 10. Since p received $f+1$ proposals for v in round k, q received at least one proposal for v in round k. Moreover, by Lemma 1, q does not receive any proposals for $1-v$ in round k. So q sets its variable x_q to v in line 12 in round k and it starts round $k+1$ with $x_q = v$. □

Lemma 3 *If a value v is k-locked for some $k \geq 1$, then every process that reaches line 12 in round k decides v in round k.*

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

Corollary 1 *If some process decides v in round $k \geq 1$, then every process that reaches line 12 in round $k+1$ decides v in round $k+1$.*

Proof By Lemmas 2 and 3. □

Corollary 2 (Uniform agreement) *If some processes p and p' decide v and v' in round $k \geq 1$ and $k' \geq 1$, respectively, then $v = v'$.*

Proof Suppose some processes p and p' decide v and v' in round $k \geq 1$ and $k' \geq 1$, respectively. There are two cases:

(1) $k = k'$. Since a process can decide a value in round k only if that value was proposed in round k, both v and v' were proposed in round k. By Lemma 1, $v = v'$.
(2) $k < k'$. Since p' decides in round k' then p' reaches line 12 in rounds $k+1, \ldots, k'$. Since p decides v in round k, by repeated applications of Corollary 1, p' decides v in rounds $k+1, \ldots, k'$. So p' decides both v and v' in the same round k'. By case (1) above, it must be that $v = v'$. □

Note that in the corollary above p and p' could be the same process.

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

Proof Suppose, for contradiction, that a process p decides a value v in some round but no process has initial value v. Then, all the processes have initial value $1-v$ and so $1-v$ is 1-locked. From Lemma 3, p decides $1-v$ in round 1. So p decides both v and $1-v$, which is a contradiction to Corollary 2. □

5.2 Termination

We now show that the algorithm satisfies the required liveness property, that is, it terminates with probability 1. We first give some intuition as to how the proof works, and then give the precise proof.

To show liveness we prove that with probability 1, some value v is k-locked for some k; therefore, by the argument of the previous section, all processes that complete round k decide v. To prove that with probability 1 some value v is k-locked, we group together pairs of adjacent rounds into epochs: an epoch r consists of rounds $2r$ and $2r+1$. For each epoch r, we consider a game that is played between the adversary and the random coins, where the adversary is trying the prevent a value from becoming locked at the end of the epoch. We show that, no matter what the adversary does in the epoch, there is a "lucky" choice of coin tosses that will foil the adversary, causing a value to be locked at the end of the epoch. Since there are at most $2n$ coin tosses in an epoch, the probability that all coin tosses are lucky is at least 2^{-2n}. Since the game is played repeatedly forever, the probability that in some epoch all the coin tosses are lucky is 1.

We now give an intuition of how to define the lucky coin tosses, that is, the coin tosses that cause a value to be locked at the end of the epoch. Note that each process changes its variable x to some value $v \in \{0, 1\}$ at the end of each round in line 12, and this can happen in two ways: v can be obtained from a coin toss, or it can be a value proposed in the round. In the first case, we say that the process R-gets v; in the second, the process D-gets v. We say that a value v is k-major at a time t if, by time t, a majority of processes have started round k with their variable x set to v. It is easy to show that if a value v is k-major at some time, then v is the only value that a process can D-get in round k. To define the lucky coin tosses, note that processes can toss coins in a round k only after the time when some process has first received $n-f$ proposals in round k; this time is denoted τ_k. The lucky coins of epoch r (i.e., of rounds k and $k+1$ for $k = 2r$) depend on what happens at times τ_k and τ_{k+1}, as follows:

- *Case 1: at time τ_k, some value v is k-major.* We define the lucky coins of epoch r to be all v. Thus, in round k, if a process D-gets a value, it D-gets v and if it R-gets a value and the coins are lucky, it R-gets v as well, causing v to be locked at the end of round k.
- *Case 2: at time τ_k, no value is k-major.* We define the lucky coins of epoch r to be 0 until time τ_{k+1} (before any process tosses a coin in round $k+1$). The subsequent lucky coins, i.e., those that occur in epoch r after time τ_{k+1}, depend on what happens at time τ_{k+1}, as follows:
  - *Case 2.1: at time τ_{k+1}, 0 is $(k+1)$-major.* Then the subsequent lucky coins of epoch r are still defined to

be 0. Thus, in round $k+1$, if a process D-gets a value, it D-gets 0 and if it R-gets a value and the coins are lucky, it R-gets 0 as well, causing 0 to be locked at the end of round $k+1$.

– *Case 2.2: at time τ_{k+1}, 0 is not $(k+1)$-major.* Then the subsequent lucky coins of epoch r are defined to be 1. In this case, we will show that, in round $k+1$, if a process D-gets a value, it D-gets 1 and if it R-gets a value and the coins are lucky, it R-gets 1 as well, causing 1 to be locked at the end of round $k+1$.

With this definition of lucky coins, if the coins are lucky, then in all cases some value is locked at the end of the epoch. We now present the detailed liveness proof.

Lemma 4 *Every correct process starts every round $k \geq 1$.*

Proof The proof is by a standard induction on the round k. Clearly, every correct process starts round $k = 1$. Suppose every correct process starts a round $k \geq 1$. Then at least $n-f$ correct processes report a value in line 5 of round k, so every correct process receives reports from at least $n-f$ processes in line 6 of round k. Thus at least $n-f$ correct processes propose a value in line 8 or 9 of round k, and every correct process receives proposals from at least $n-f$ processes in line 10 of round k. So every correct process completes its wait statement in line 10 of round k, and starts round $k+1$. □

Corollary 4 *If a value v is k-locked for some $k \geq 1$, then every correct process decides v in round k.*

Proof Immediate from Lemmas 3 and 4. □

For each $k \geq 1$, we say that a value v *is k-major at time t* if by time t more than $n/2$ processes have started round k with their variable x set to v.[4] Clearly, for each $k \geq 1$ 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 its variable x_p to v in line 12 of round k. If v was obtained from tossing a coin, we say that *p R-gets v in round k*; otherwise, we say that *p D-gets v in round k*.

Lemma 5 *For every $k \geq 1$: (1) if some process D-gets v in round 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 round k.*

Proof Consider round $k \geq 1$. Suppose p D-gets v in round k. 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 round k. Thus, v was k-major—proving part (1). Part (2) follows from part (1) and the fact that v and $1-v$ cannot both be k-major. □

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

For the rest of the proof, we group pairs of rounds into *epochs* as follows: *epoch $r \geq 1$* consists of rounds $k = 2r$ and $k+1 = 2r+1$.[5] We now 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.

For every $k \geq 1$, let τ_k be the first time that any process receives $n-f$ proposals in round k. From Lemma 4, for every $k \geq 1$, some process receives $n-f$ proposals in round k, and so τ_k is well-defined. Note that no process tosses a coin in round k before time τ_k.

We say that *epoch r is lucky* if, for every process p and every time t, if p tosses a coin in epoch r at time t, then p gets the value $\mathit{FavorableToss}(r, t)$ from the coin, where $\mathit{FavorableToss}(r, t)$ is the function defined in Fig. 2. Note that if p tosses a coin in epoch r at time t, this occurs after at least one process receives $n-f$ proposals in round $k = 2r$, and so this occurs at time $t \geq \tau_k$. From the code defining the function $\mathit{FavorableToss}(r, t)$, it is clear that to evaluate $\mathit{FavorableToss}(r, t)$ at any time $t \geq \tau_k$, one only needs to look at events that occur in the system up to time t. So whenever we need to evaluate the function $\mathit{FavorableToss}(r, t)$ to determine whether a particular coin toss was lucky or not, we only need to see what happened in the system before this coin toss occurs.

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 rounds k and $k+1$. Since epoch r is lucky, if any process R-gets a value v at some time t and in round k or $k+1$, then $v = \mathit{FavorableToss}(r, t)$ 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 t such that $\tau_k \leq t$, $\mathit{FavorableToss}(r, t) = v$. So, v is the only value that a process can R-get in round k. Since v is k-major, by Lemma 5, v is also the only value that a process can D-get in round k. Thus, every process that completes round k does so with its variable x set to v. So, 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 t such that $\tau_{k+1} \leq t$, $\mathit{FavorableToss}(r, t) = 0$. So, 0 is the only value that a process can R-get in round $k+1$. Since 0 is $(k+1)$-major, by Lemma 5, 0 is also the only value that a process can D-get in round $k+1$. Thus, 0 is $(k+2)$-locked.

[5] Round 1 is not part of any epoch.

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

  $k \leftarrow 2r$
  Case A ($\tau_k \leq t$ and some value $v$ is $k$-major at time $\tau_k$): return $v$
  Case B ($\tau_k \leq t$ and no value is $k$-major at time $\tau_k$):
    Subcase B.1 ($\tau_k \leq t < \tau_{k+1}$): return 0
    Subcase B.2 ($\tau_{k+1} \leq t$ and 0 is $(k+1)$-major at time $\tau_{k+1}$): return 0
    Subcase B.3 ($\tau_{k+1} \leq t$ and 0 is not $(k+1)$-major at $\tau_{k+1}$): return 1
```

Fig. 2 Favorable coin toss algorithm

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 at least one process p starts round $k+1$ with its variable x_p set to 1; thus either p D-got 1, or R-got 1 in round k at some time $\tau_k \leq t < \tau_{k+1}$. Since no value is k-major at time τ_k, for $\tau_k \leq t < \tau_{k+1}$ we have $FavorableToss(r, t) = 0$, and so p D-got 1 in round k. Thus, by Lemma 5, 1 was k-major at some time. Since 1 was k-major, by Lemma 5, 1 is the only value that a process can D-get in round k. Moreover, for $\tau_{k+1} \leq t$, we have $FavorableToss(r, t) = 1$, and so 1 is the only value that a process can R-get in round k at or after time τ_{k+1}. Thus, from time τ_{k+1} on, 1 is the only value that a process can D-get or R-get in round k. So, from time τ_{k+1} on, no process starts round $k+1$ with its variable x 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 round $k+1$. Moreover, for $\tau_{k+1} \leq t$, we have $FavorableToss(r, t) = 1$, and so 1 is the only value that a process can R-get in round $k+1$. Thus, 1 is $(k+2)$-locked. □

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 round there are at most n coin tosses), 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 rounds). □

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

Proof Immediate from Lemmas 7 and 6, and Corollary 4. □

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

By Corollaries 2 and 3, Ben-Or's algorithm satisfies the safety properties of consensus, and by Lemma 8 it satisfies the liveness property, so we have the following:

Theorem 1 *Ben-Or's algorithm solves consensus under a strong adversary for any $f < n/2$.*

6 Global-coin variant of Ben-Or's algorithm

In Ben-Or's algorithm, if a process receives proposals only for ? in a round k, it tosses an independent coin and changes its estimate to the random bit that is the outcome of that coin. The hope is that every process that completes round k will change its estimate to the same value, because this immediately triggers agreement and termination. Thus, intuitively, if all the processes that toss a coin in a round get the *same* random bit, the algorithm should continue to work and its expected time to terminate should improve dramatically.

The above intuition leads to the well-known concept of a "global coin", an abstraction that is commonly used to speed up and modularize randomized consensus algorithms. Instead of tossing independent coins, processes toss a *global coin* with parameter ρ where $0 < \rho \leq 1/2$; this is a coin such that for each possible outcome $v \in \{0, 1\}$, *with probability at least* ρ, all the processes that toss the coin get the same outcome v [7]. The strongest global coin is the one with parameter $\rho = 1/2$: *all* the processes that toss this coin are guaranteed to get the *same* random bit, i.e., they all get 0 or they all get 1, each case with probability 1/2.[6]

In this paper, we consider the global-coin variant of Ben-Or's algorithm where, in each round, processes use the global coin with parameter $\rho = 1/2$. More precisely, the global-coin variant of Ben-Or's algorithm is obtained by replacing line 12 of Ben-Or's algorithm in Fig. 1 with the following line:

[6] Note that this global coin with parameter 1/2 also satisfies the specification of a global coin with parameter ρ for every ρ (where $0 < \rho \leq 1/2$).

```
if received at least one $(P, k, v)$ with $v \neq ?$ then $x \leftarrow v$
else $x \leftarrow$ global-coin$(k)$ {get global coin of round $k$}
```

where, for each round $k \geq 1$ and each $u \in \{0, 1\}$, with probability 1/2, *global-coin*$(k) = u$ at all processes.

In the next section we show that, contrary to the intuition mentioned above, the use of this strong global coin does not speed up Ben-Or's algorithm; in fact it may do the opposite: with probability 1/2, the algorithm does not terminate.

7 The global-coin variant of Ben-Or's algorithm does not terminate

We now show that, when $n/3 \leq f < n/2$, the global-coin variant of Ben-Or's algorithm does not work under a strong adversary. To do so, we describe how a strong adversary can schedule the process steps and message receipts in a way that prevents processes from ever deciding. To achieve its goal, however, the adversary needs to be lucky in the first round of the algorithm: as we explain below, the value of the global coin in round 1 has to be favorable for the adversary's strategy to work; this occurs with probability 1/2. If the adversary is lucky in round 1, then it can prevent processes from deciding in every subsequent round *irrespective of the outcomes of the global coin in those rounds*. In other words, with probability 1/2, the adversary can prevent the algorithm's termination.

Theorem 2 *The global-coin variant of Ben-Or's algorithm does not solve consensus under a strong adversary for any f such that $n/3 \leq f < n/2$.*

Proof Consider a system where $2f + 1 \leq n \leq 3f$. We show that, under a strong adversary, the global-coin variant of Ben-Or's algorithm does not satisfy the liveness property of consensus, namely, termination with probability 1. We first explain the basic idea of the proof (for the special case of $n = 3$ and $f = 1$), and then give the actual proof (for the general case where $2f + 1 \leq n \leq 3f$).

Consider a system with $n = 3$ processes with initial values 0, 1, 1, respectively, and $f = 1$. We now outline how, with probability 1/2, a strong adversary can prevent processes in this system from ever deciding. The basic idea is that for each round $k \geq 2$, the adversary causes two processes to start round k with opposite estimates, and holds the third process in round $k - 1$ until it learns the value C_k of the global coin of round k. To learn C_k, the adversary schedules the steps of the two processes that start round k with opposite estimates until one of them tosses the (global) coin in round k. After it learns C_k, the adversary causes the process that was held in round $k - 1$ to complete round $k - 1$ and start round k with estimate $1 - C_k$. So two (i.e., a majority of) processes start round k with estimate $1 - C_k$. Now the adversary has the ability to cause a process to complete round k with either estimate 0 or estimate 1—as the adversary chooses. To see this, note that in round k, if a process obtains its estimate from the global coin, it obtains C_k; otherwise, the process obtains the value $1 - C_k$, which is the value with which a majority of processes started round k. Thus, for every $k \geq 2$, the adversary can cause two processes to complete round k and start round $k+1$ with two opposite estimates, while holding a third process in round k with the ability to control the estimate with which this process will later complete round k. To bootstrap this scheme, however, the adversary needs a "lucky" initial round $k = 1$, as follows.

The three processes start round 1 with estimates 0, 1, 1, respectively (their initial values). The adversary is lucky if the outcome C_1 of the global coin of round 1 is 0, which happens with probability 1/2. If the adversary is lucky, it can cause two processes to start round 2 with opposite estimates, while the third process is held in round 1 and could finish this round with either estimate 0 or 1, as the adversary can later choose (the argument is similar to the one given above). By collating this "lucky" round 1 with the rounds $k \geq 2$ described above, we see that, with probability 1/2, this adversary can cause processes to never decide.

We now proceed with the actual proof of the theorem. Let n and f be such that $2f + 1 \leq n \leq 3f$. Consider a system with n processes $p_1, p_2, \ldots, p_n$, and partition these processes in three groups of $f, f, n - 2f$ processes, respectively. This partition is fixed throughout the proof, and a group will always refer to one of these three groups in the partition. Note that a single group has at most f processes, since $n - 2f \leq f$. Moreover, any two groups together contain at least $n - f$ processes, so the algorithm must make progress even if the third group crashes or its messages are slow to arrive. In the run that we construct, all processes of a group start with the same initial value and execute the same steps, so they all go through the same state transitions. We shall refer to the estimate and initial value of the group to mean the estimate or initial value held by all processes in the group; similarly, we shall say that the group performs an action (such as receiving certain messages) to mean that all processes of the group perform that action.

Suppose that one group starts the consensus algorithm with initial value 0, and the other two groups start with initial value 1.

Let C be any infinite sequence of global coin tosses *that starts with coin toss 0*, i.e., $C \in 0 \cdot \{0, 1\}^*$; the k-th element of C, denoted C_k, is the value of the coin toss in round k of the algorithm. We show below that for every such C, the adversary can generate an infinite run R_C where no process ever decides. In R_C, the adversary's strategy depends on the coins in C, but only those coins that were already tossed when the adversary takes an action. In other words, the strategy does not depend on the value of future coins.

To describe R_C, we consider an infinite sequence of consecutive time periods T_k, for $k \geq 1$, and describe what happens in run R_C in each one of these periods. We will show that at the end of each period T_k the following holds:

1. Two groups are at the start of round $k+1$; one group has estimate 0 and the other has estimate 1; neither one has decided.
2. The third group is still in round k waiting to receive proposals. Furthermore, the adversary has the ability to cause the group to later complete round k with either estimate 0 or 1—as the adversary chooses—without deciding.

Period T_1. At the beginning of period T_1, the three groups are at the start of round 1, where one group has estimate 0, while the other two groups have estimate 1. Let G_0 be the group with estimate 0 and G_1 and G_2 be the other two groups, which have estimate 1. During period T_1, the adversary schedules the steps of the processes and the receipt of messages as follows.

Groups G_0 and G_1 send reports for 0 and 1, respectively, for round 1. Then each group receives those reports from both groups, so they receive at least $n - f$ reports (recall that two groups combined contain at least $n - f$ processes). Since each group has at most f processes and $f \leq n/2$, both groups send proposals for ?.

Now group G_0 receives these proposals for ? from G_0 and G_1. Since every received proposal is for ?, group G_0 tosses a coin and changes its estimate to the value of the global coin of round 1, i.e., group G_0 changes its estimate to $C_1 = 0$. Thus, group G_0 completes round 1 with estimate 0 without deciding; in the meantime, group G_1 is still waiting to receive proposals in round 1.

Then the third group G_2 proceeds as follows: (a) it sends reports for 1, (b) it receives reports for 1 (from groups G_1 and G_2), and (c) since it receives at least $n - f > n/2$ reports for 1, it sends proposals for 1.

Next, group G_2 continues as follows: (a) it receives the proposals for 1 sent by G_2 and the proposals for ? sent by G_0; (b) it changes its estimate to 1; and (c) it completes round 1 without deciding (since it received at most f proposals for a value other than ?). Note that group G_1 is still waiting to receive proposals in round 1.

At this point, which is the end of time period T_1, the situation is as follows:

1. Two groups have completed round 1 and are at the start of round 2; one of them has estimate 0 and the other has estimate 1; neither one has decided. These groups are G_0 and G_2.
2. The third group is still in round 1 waiting to receive proposals. Furthermore, the adversary has the ability to cause this group to later complete round 1 with either estimate 0 or 1—as the adversary chooses—without deciding.
   To see this, note that in the future the adversary can schedule group G_1 to receive the proposals for ? from groups G_0 and G_1 in round 1, in which case group G_1 tosses a coin and terminates round 1 with estimate $C_1 = 0$; or the adversary can schedule group G_1 receive the proposals for 1 from G_2 and the proposals for ? from G_1, in which case group G_1 terminates round 1 with estimate 1; in either case, group G_1 terminates round 1 without deciding (since it received at most f proposals for a value other than ?).

This concludes the description of the initial time period T_1 of the run R_C.

Period T_k **for** $k \geq 2$. At the beginning of period T_k, the state is the same as at the end of period T_{k-1}, namely:

1. Two groups are at the start of round k; one of them has estimate 0 and the other has estimate 1; neither one has decided.
2. The third group is still in round $k - 1$ waiting to receive proposals. Furthermore, the adversary has the ability to cause the group to later complete round $k - 1$ with either estimate 0 or 1—as the adversary chooses—without deciding.

Let G_0 and G_1 be the two group of processes which are at the start of round k with estimates 0 and 1, respectively, and G_D (the subscript D is for "delayed") denotes the third group, which is still in round $k-1$, at the beginning of period T_k, as described above.

During period T_k, the adversary schedules the steps of the processes and the receipt of messages as follows. Groups G_0 and G_1 send reports for 0 and 1, respectively, for round k. Then each group receives those reports from both groups. Since each group has at most f processes and $f \leq n/2$, both groups send proposals for ?. Now group G_0 receives these proposals for ? from both groups, and it tosses a coin and changes its estimate to the value of the global coin of round k, namely C_k. Thus, group G_0 completes round k with estimate C_k without deciding; in the meantime, group G_1 is still waiting to receive proposals in round k.

The adversary now knows C_k and it uses this information to schedule some process steps and message receipts. Specifically, it first causes group G_D to complete round $k - 1$ with the opposite estimate $1 - C_k$ without deciding. Then, group G_D continues its execution in round k, as follows: (a) it sends reports for $1 - C_k$, (b) it receives reports for $1 - C_k$ from G_D and from either G_0 (if $1 - C_k = 0$) or G_1 (if $1 - C_k = 1$), (c) it sends proposals for $1 - C_k$, since it received at least $n - f > n/2$ reports for $1 - C_k$.

Now, group G_D proceeds as follows: (a) it receives the proposals for $1 - C_k$ from G_D and the proposals for ? from G_0, (b) it changes its estimate to $1 - C_k$, and (c) it completes round k without deciding (since it received at most f proposals for a value other than ?). Note that group G_1 is still waiting to receive proposals in round k.

At this point, which is the end of time period T_k, the state is as follows:

1. Two groups are at the start of round $k + 1$; one of them has estimate 0, and the other has estimate 1; neither has decided. These two groups are G_0 and G_D: G_0 completes round k with estimate C_k, while G_D completes round k with estimate $1 - C_k$.
2. The third group is still in round k waiting to receive proposals. Furthermore, the adversary has the ability to cause this group to later complete round k with either estimate 0 or 1—as the adversary chooses—without deciding.
   To see this, note that the adversary can schedule group G_1 to receive the proposals for ? from G_0 and G_1 in round k, in which case group G_1 tosses a coin and terminates round k with estimate C_k; or the adversary can schedule group G_1 to receive the proposals for $1 - C_k$ from G_D and the proposals ? from G_1, in which case group G_1 terminates round k with estimate $1 - C_k$; in either case, group G_1 terminates round k without deciding (since it received at most f proposals for a value other than ?).

This concludes the description of time period T_k of the run R_C, for $k \geq 2$.

Run R_C is defined by collating the successive time periods T_k, for $k = 1, 2, \ldots$, that we defined above. Note that in run R_C no process ever decides.

We have just shown that in a system with n processes, where $2f+1 \leq n \leq 3f$, there is a strong adversary such that, if f processes start with 0 and $n - f$ processes start with 1, then for every sequence of global coin tosses C that start with 0, the adversary can generate a run R_C where no process decides. Since the probability of such a C occurring is 1/2, then, starting from this initial state, this adversary can cause the algorithm to never terminate with probability at least 1/2. □

Theorem 3 *The global-coin variant of Ben-Or's algorithm solves consensus under a strong adversary if and only if* $f < n/3$.

Proof

1. If $f < n/3$, then the global-coin variant of Ben-Or's algorithm works against a strong adversary: the proof is very similar to the one of Ben-Or's algorithm given in [14, Chapter 21.3] for the case that $f < n/3$.
2. If $f \geq n/3$, then either $f \geq n/2$, in which case if f processes initially crash then correct processes never decide as they cannot receive the $f + 1$ required proposals in line 11; or $n/3 \leq f < n/2$, in which case by Theorem 2 the algorithm does not solve consensus (because it also does not satisfy the required liveness property). □

It is worth mentioning that the global-coin variant of Ben-Or's algorithm does work for $f < n/2$ under a *weak* adversary. The proof is simple and similar to the one in [13].

8 Recent work in randomized consensus

Since Ben-Or's algorithm was published in 1983 dozens of papers have been published on randomized consensus, in both message-passing and shared-memory systems, for various types of adversaries, failure modes, cryptographic assumptions, etc. Much of this work focuses on improving the complexity of the algorithms in terms of the expected total or individual work required, or providing related lower bounds. In particular, recent papers by Attiya and Censor [1] and by Aspnes [5] give the first randomized consensus algorithms for shared-memory systems that achieve *optimal* $O(n^2)$ total expected work (the algorithm in [1] uses multi-writer registers while the one in [5] uses only single-writer registers). These papers also contain excellent surveys of the progress made in this area since the 1990 randomized shared-memory algorithm by Bracha and Rachman [9], and they include a comprehensive set of relevant references. A related paper by Attiya and Censor derives lower bounds that quantify the trade-off between the probability of termination and the total step complexity of randomized consensus algorithms [2]. These lower bounds apply to both shared-memory and message-passing systems. Finally, a good survey of randomized consensus protocols can be found in [4].

9 Concluding remarks

We presented the first correctness proof for Ben-Or's pioneering randomized consensus for the case that $f < n/2$ process crashes and the adversary is strong. Even though the algorithm is quite simple, its proof of termination with probability 1 is unexpectedly complex: it requires reasoning over two consecutive rounds at a time, and it exposes some subtle issues due to the power of the strong adversary in a message-passing system. We also proved that the global-coin variant of Ben-Or's algorithm is incorrect under an ideal global coin, which ensures that all processes obtain the same random bit. This shows that, contrary to a common belief, replacing independent coins with an ideal global coin (in a randomized consensus algorithm) can sometimes be deleterious rather than beneficial: instead of speeding up the algorithm, the use of an ideal global coin may actually prevent termination.

What if the global coin is not ideal, that is, what if there is a small probability ϵ that different processes obtain different random bits? Using a similar construction as the one in Sect. 7, it is possible to show that Ben-Or's algorithm with such global coins has a lower bound of $\Omega(1/\epsilon)$ expected rounds for termination. Note that the closer the coin is to an ideal global coin, the closer ϵ approaches 0, and the larger the lower bound on the expected number of rounds for termination. In particular, if $\epsilon \to 0$ as $n \to \infty$, for example $\epsilon \in \Theta(1/n)$, then the global-coin variant of Ben-Or's algorithm does not terminate in a constant expected number of rounds.

Note that the common technique to replace independent coins with a global coin to speed up randomized algorithms, which does not work with Ben-Or's algorithm, works with the well-known consensus algorithm of Aspnes and Herlihy [3].[7] This begs the question of why that is the case. Roughly speaking, in contrast to Ben-Or's algorithm, in the algorithm of [3] the value that processes can obtain deterministically in a round is fixed before the global coin of that round is revealed. We call this property *early binding* of the deterministic value of a round. With early binding, it is impossible for the adversary to first learn the global coin of a round and then choose a different value for processes to obtain deterministically in that round. More precisely, with this property the adversary cannot (a) first estimate the global coin of a round by observing the outcome of the coin tosses of some processes, and then (b) schedule other processes such that they deterministically obtain a value that is different from this estimate. We believe that the early binding property is important for the use of global coins with a strong adversary.

Finally, we note that a strong adversary in message-passing systems seems to have more power than in shared-memory systems. In particular, when a process sends a value to all processes in an asynchronous message-passing system, the adversary can delay the receipt of some of these messages for any finite period of time. In contrast, in a shared-memory system, once a process writes a value in a shared register, every other process can see it. It is also worth noting that a recent work discovered some subtle issues regarding adversaries in systems with shared objects: this work shows some previously overlooked relation between the power of adversaries and different definitions of linearizability of the implemented shared objects [12].

[7] This algorithm was used to obtain the optimal randomized consensus algorithms in [1,5] by plugging better implementations of the global coin.

Acknowledgments We would like to thank Roberto Segala for his comments on a preliminary version of this paper [6], and the anonymous reviewers for their helpful comments. This work was partially supported by the National Science and Engineering Research Council of Canada.

References

1. Attiya, H., Censor, K.: Tight bounds for asynchronous randomized consensus. J. ACM **55**(5), 20:1–20:26 (2008)
2. Attiya, H., Censor-Hillel, K.: Lower bounds for randomized consensus under a weak adversary. SIAM J. Comput. **39**(8), 3885–3904 (2010)
3. Aspnes, J., Herlihy, M.: Fast randomized consensus using shared memory. Journal of Algorithms **11**, 441–461 (1990)
4. Aspnes, J.: Randomized protocols for asynchronous consensus. Distrib. Comput. **16**(2–3), 165–175 (2003)
5. Aspnes, J.: Randomized consensus in expected $O(n^2)$ total work using single-writer registers. In: International Symposium on Distributed Computing, pp. 263–273 (2011)
6. Aguilera, M.K., Toueg, S.: Correctness proof of Ben-Or's randomized consensus algorithm. Technical report TR98-1682, Department of Computer Science, Cornell University, Ithaca, NY 14853 (1998)
7. Attiya, H., Welch, J.: Distributed Computing, 2nd edn. Wiley, Hoboken (2004)
8. Ben-Or, M.: Another advantage of free choice: completely asynchronous agreement protocols. In: ACM Symposium on Principles of Distributed Computing, pp. 27–30 (1983)
9. Bracha, G., Rachman, O.: Randomized consensus in expected $O(n^2 \log n)$ operations. In: International Workshop on Distributed Algorithms, pp. 143–150 (1991)
10. Chor, B., Dwork, C.: Randomization in Byzantine agreement. Adv. Comput. Res. **4**, 443–497 (1989)
11. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985)
12. Golab, W.M., Higham, L., Woelfel, P.: Linearizable implementations do not suffice for randomized distributed computation. In: ACM Symposium on Theory of Computing, pp. 373–382 (2011)
13. Hadzilacos, V.: Lecture notes. Unpublished manuscript (1991)
14. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers, Inc, San Fransisco (1996)
15. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. **22**, 299–319 (1990)