LOWER BOUNDS FOR RANDOMIZED CONSENSUS UNDER A WEAK ADVERSARY∗

HAGIT ATTIYA† AND KEREN CENSOR-HILLEL‡

Abstract. This paper studies the inherent trade-off between termination probability and total step complexity of randomized consensus algorithms. It shows that for every integer \(k\), the probability that an \(f\)-resilient randomized consensus algorithm of \(n\) processes does not terminate with agreement within \(k(n - f)\) steps is at least \(\frac{1}{c}\), for some constant \(c\). A corresponding result is proved for Monte-Carlo algorithms that may terminate in disagreement. The lower bound holds for asynchronous systems, where processes communicate either by message passing or through shared memory, under a very weak adversary that determines the schedule in advance, without observing the algorithm’s actions. This complements algorithms of Kapron et al. [Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), ACM, New York, SIAM, Philadelphia, 2008, pp. 1038–1047] for message-passing systems, and of Aumann [Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC), ACM, New York, 1997, pp. 209–218] and Aumann and Bender [Distrib. Comput., 17 (2005), pp. 191–207] for shared-memory systems.

Key words. distributed computing, shared memory, message passing, lower bound, randomized algorithms, consensus

AMS subject classifications. 68Q85, 68Q87, 68W15, 68W20

DOI. 10.1137/090751906

1. Introduction. At the heart of many coordination problems in distributed systems lies the need to reach consensus among processes, despite the possibility of process failures. A (binary) consensus algorithm allows processes starting with input values in \(\{0, 1\}\) to agree on the same output value (agreement). To rule out trivial solutions, this common output must be one of the inputs (validity), and every process must eventually decide (termination). It is well known that no deterministic algorithm can achieve consensus in an asynchronous system if one process may fail [23, 25, 31]. One successful approach for circumventing this impossibility result is to employ randomization and relax the termination property to hold with high probability. In typical randomized algorithms for consensus, the probability of not terminating in agreement decreases as the execution progresses, becoming smaller as processes perform more steps.

This paper shows that this behavior is inherent, by proving lower bounds on the probability of termination when the step complexity is bounded. In order to make the lower bounds as strong as possible, we assume a very weak adversarial model, which fixes the complete schedule in advance, without observing the steps of the algorithm. In particular, the schedule is determined without knowing results of local coin-flips, contents of messages sent, or memory locations accessed.

∗Received by the editors March 9, 2009; accepted for publication (in revised form) October 2, 2010; published electronically December 22, 2010. A preliminary version of this paper appeared in Proceedings of the 27th Annual ACM Symposium on Principles of Distributed Computing, 2008, pp. 315–324. This research was supported in part by the Israel Science Foundation (grant 953/06).
†Department of Computer Science, Technion, Haifa 32000, Israel (hagit@cs.technion.ac.il).
‡Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA 02139 (ckeren@csail.mit.edu). This author’s research was supported by the Simons Postdoctoral Fellows Program. This work was done while this author was a Ph.D. student at the Department of Computer Science, Technion, Haifa 32000, Israel and supported in part by the Adams Fellowship Program of the Israel Academy of Sciences and Humanities.
We prove that for every integer $k$, the probability that an $f$-resilient randomized consensus algorithm of $n$ processes does not terminate after $k(n - f)$ steps is at least $\frac{1}{c^k}$, where $c$ is a constant if $\lceil \frac{n}{f} \rceil$ is a constant. The result holds for asynchronous message-passing systems and asynchronous shared-memory systems (using reads and writes), albeit with different constants. While the same general proof structure applies in both cases, it is accomplished differently in the message-passing and the shared-memory models; the latter case is further complicated due to the adversary’s weakness.

For the message-passing model, our proof extends and improves on a result of Chor, Merritt, and Shmoys [20] for synchronous message-passing systems. They show that the probability that a randomized consensus algorithm does not terminate after $k$ rounds (and $k(n - f)$ steps) is at least $\frac{1}{c^k}$. (A similar result is attributed to Karlin and Yao [28].) The proof rests on considering a specific chain of indistinguishable executions and showing a correlation between the termination probability and the length of this chain,\(^1\) which in turn depends on the number of rounds. The chain is taken from the proof of the rounds’ lower bound for (deterministic) consensus [21, 22] (cf. [6, Chapter 5]); since the chain is determined in advance, i.e., regardless of the algorithm’s transitions, the lower bound is derived with a weak adversary. (An overview of the proof strategy appears in section 2.2.)

Our first contribution, for the message-passing model, improves on this lower bound by exploiting the fact that asynchrony allows us to construct “shorter” indistinguishability chains. This shows that the probability that an asynchronous randomized consensus algorithm does not terminate after $k(n - f)$ steps is at least $\frac{1}{c^k}$, where $c$ is a constant if $\lceil \frac{n}{f} \rceil$ is a constant. (The lower bound for asynchronous message-passing systems appears in section 3.)

Substituting specific values in our lower bound implies that any randomized consensus algorithm has probability at least $\frac{1}{\text{polylog}(n)}$ for not terminating within $\log \log n$ (asynchronous) rounds, and probability at least $\frac{1}{\text{poly}(n)}$ for not terminating within $\log n$ (asynchronous) rounds.

The lower bound can be extended to Monte-Carlo algorithms that always terminate, at the cost of compromising the agreement property (section 5). If an asynchronous message-passing algorithm always terminates within $k(n - f)$ steps, then the probability for disagreement is at least $\frac{1}{c^k}$, where $c$ is a constant if $\lceil \frac{n}{f} \rceil$ is a constant. This lower bound can be compared to the recent consensus algorithms of Kapron et al. [27] for the message-passing model. One algorithm always terminates within $\text{polylog}(n)$ asynchronous rounds, and has a probability $\frac{1}{\text{polylog}(n)}$ for disagreeing, while the other terminates within $2^{9(\log^7 n)}$ asynchronous rounds, and has a probability $\frac{1}{\text{poly}(n)}$ for disagreeing.

There is an extensive amount of literature on randomized agreement algorithms for message-passing systems. Recent papers in this area provide algorithms for agreement in the presence of Byzantine processes in full information models, where the adversary is computationally unbounded. See [11, 24, 29] for a more detailed description and references.

In principle, the lower bound scheme can be extended to the shared-memory model by focusing on layered executions [5, 32]. However, our severely handicapped adversarial model poses a couple of technical challenges. First, while in the message-passing model each step can be assumed to send messages to all processes, in a shared-memory event a process chooses which register to access and whether to read from it\(^1\) The length of the chain is the number of executions in it.
or write to it. A very weak adversary, as we use for our lower bounds, must find a way to make its scheduling decisions in advance without even knowing what type of step the process will take. Second, the proof scheme requires schedules to be determined independently of the coin-flips. The latter difficulty cannot be alleviated even by assuming a stronger adaptive adversary that may schedule the processes according to the execution so far.

We manage to extend the lower bound scheme to the shared-memory model, by first simplifying the model, assuming that processes either write to single-writer registers or perform a cheap-snapshot operation, reading all the registers at once. By further assuming that an algorithm regularly alternates between writes and cheap snapshots, we make processes' steps predictable, allowing a weak adversary to construct indistinguishability chains. The lower bound is extended to hold for multiwriter registers by reduction; while ordinary simulations of multiwriter registers using single-writer registers have $O(n)$ overhead (which would nullify the lower bound), cheap snapshots admit a simulation with constant overhead. (The results for the shared-memory model appear in section 4.)

The lower bounds we obtain for the shared-memory model are the same as for the message-passing model, though with different constants. (More detailed calculations and comparison with related work appear in section 6.)

To the best of our knowledge, there are no other lower bounds on randomized consensus in shared-memory systems under a weak adversary. There are several algorithms assuming a value-oblivious adversary, which may determine the schedule adaptively based on the functional description of past and pending operations, but cannot observe any value of any register or results of local coin-flips. This model is clearly stronger than the adversary we employ, and hence our lower bounds apply to it as well.

The algorithms differ by the type of shared registers they use [7, 8, 9, 15]. For single-writer multireader registers, Aumann and Bender [8] give a consensus algorithm that has probability of at most $1/n$ of not terminating within $O(n \log^2 n)$ steps. For multiwriter multireader registers, Aumann [7] shows a consensus algorithm in which the probability of not terminating in $k$ iterations (and $O(k \cdot n)$ steps) is at most $(3/4)^k$.

Chandra [15] gives an algorithm with $O(\log^2 n)$ individual step complexity, assuming an intermediate adversary that cannot see the outcome of a coin-flip until it is read by some process. Aumann and Kapah-Levy [9] give an algorithm with $O(n \log n \exp(2\sqrt{\text{polylog}n}))$ total step complexity, using single-writer single-reader registers, and assuming a value-oblivious adversary.

An algorithm with $O(n \log \log n)$ total step complexity against a weak adversary was given by Cheung [18], which considers a model with a stronger assumption that a write operation occurs atomically after a local coin-flip. It improves upon earlier work by Chor, Israeli, and Li [19], who provide an algorithm with $O(n^2)$ total step complexity using a slightly different atomicity assumption.

Other related work on randomized consensus assumes strong adversaries, which adapt to the computation, scheduling processes dynamically, after observing the results of their local coin-flips. A great deal of study was invested, yielding numerous algorithms (see the survey in [2]). Lower bounds were given on the expected number of coin-flips [1], on the expected number of rounds in synchronous systems [10], and a tight $\Theta(n^2)$ bound on the total step complexity in asynchronous systems [5].
2. The lower bound strategy.

2.1. The model in brief. We consider a standard model of an asynchronous system with a set of \( n \geq 3 \) processes \( P = \{ p_1, \ldots, p_n \} \). Each step of a process consists of some local computation, including an arbitrary number of coin-flips (possibly biased), and a communication operation, which depends on the communication model.

In a message-passing system processes communicate by sending and receiving messages: the communication operation of a process is sending messages to some subset of the processes and receiving messages from some subset of them. For the lower bounds, we assume that a process sends a message to all the processes in each step. In a shared memory system, processes communicate by reading and writing to shared registers; each step of a process is either a read or a write to some register.

The local coin-flips of a process \( p_i \) are modeled as a random string of coin-flip results \( c_i \), which can be accessed by \( p_i \) but are unavailable to any other process.

All of the randomization of the algorithm is encompassed within \( n \) coin-flip strings \( \vec{c} = (c_1, \ldots, c_n) \).

Additional nondeterminism is introduced by the scheduling choices made by an adversary. We assume a weak adversary that is nonadaptive and decides on the scheduling in advance. The adversary does not observe the results of any local coins a process flips or any operation a process performs.

A schedule \( \sigma \), together with an initial configuration \( I \) and \( n \) coin-flip strings \( \vec{c} = (c_1, \ldots, c_n) \), determines an execution \( \alpha(\sigma, \vec{c}, I) \).

2.2. The lower bound approach. Let \( A \) be an \( f \)-resilient asynchronous randomized consensus algorithm. Let \( q_k \) denote the maximum probability, over all weak adversaries and over all initial configurations, that \( A \) does not terminate after a total of \( k(n - f) \) steps are taken.

In order to prove a lower bound on \( q_k \), we consider a restricted set of schedules that proceed in layers [5, 32]. An \( f \)-layer is a sequence of at least \( n - f \) distinct process id’s. When executing a layer \( L \), each process \( p \in L \) takes a step, in the order specified by the layer.

We will consider only schedules that are \( f \)-schedules. A schedule \( \sigma = L_1, L_2, \ldots, L_k \) is an \( f \)-schedule if it is a finite sequence of \( f \)-layers. A process \( p_i \) is nonfaulty in layer \( r \) if it appears in the layer. A process \( p_i \) crashes in layer \( r \) if it does not take a step in any layer \( \ell \geq r \). A process is skipped in layer \( r \) if it does not appear in layer \( r \) but appears in one of the following layers.

**Definition 2.1.** For a schedule \( \sigma \), let \( \text{crash}(\sigma, p, r) \) be the schedule that is the same as \( \sigma \), except that \( p \) crashes in layer \( r \), i.e., does not take a step in any layer \( \ell \geq r \). For a set \( P \) of processes, \( \text{crash}(\sigma, P, r) \) is defined similarly.

As mentioned in the introduction, our proof will make use of indistinguishable executions. Intuitively, two finite executions \( \alpha \) and \( \alpha' \) are indistinguishable to a process \( p \) if it cannot tell the difference between them. This implies that \( p \) terminates in \( \alpha \) with a decision value \( v \) if and only if it terminates in \( \alpha' \) with a decision value \( v \). The formal definition of indistinguishability is model-dependent and will be given separately in sections 3 and 4, but we proceed formally to define indistinguishability chains as follows.

Given two executions \( \alpha_1 \) and \( \alpha_2 \) with the same \( n \) coin-flip strings \( \vec{c} = (c_1, \ldots, c_n) \), we denote \( \alpha_1 \overset{p_i}{\sim} \alpha_2 \) if process \( p_i \) does not distinguish between \( \alpha_1 \) and \( \alpha_2 \), and does not crash in them. In this case, \( p_i \) decides on the same value in \( \alpha_1 \) and in \( \alpha_2 \). We
\[
\begin{align*}
\beta_1 &= \alpha(\sigma_{\text{full}}, \vec{c}, C_0) & \text{decision is 0}
\end{align*}
\]
\[
\begin{align*}
\beta_2 &
\vdots
\end{align*}
\]
\[
\begin{align*}
\beta_m &
\vdots
\end{align*}
\]
\[
\begin{align*}
\beta_{m+1} &= \alpha(\sigma_{\text{full}}, \vec{c}, C_S) & \text{decision is 1}
\end{align*}
\]
\[
\begin{align*}
\beta_2 &
\vdots
\end{align*}
\]
\[
\begin{align*}
\beta_m &
\vdots
\end{align*}
\]
\[
\begin{align*}
\beta_{m+1} &= \alpha(\sigma_{\text{full}}, \vec{c}, C_S) & \text{decision is 1}
\end{align*}
\]

\text{Fig. 2.1. Illustration for the proof of Theorem 2.2.}

denote \(\alpha_1 \approx_m \alpha_2\) if there is a chain of executions \(\beta_1, \ldots, \beta_{m+1}\) such that
\[
\alpha_1 = \beta_1 \overset{p_{\beta_1}}{\sim} \beta_2 \overset{p_{\beta_2}}{\sim} \cdots \overset{p_{\beta_m}}{\sim} \beta_{m+1} = \alpha_2.
\]
We call such a chain an \textit{indistinguishability chain}. Clearly, if \(\alpha \approx_m \beta \approx_m \gamma\), then \(\alpha \approx_{m+n} \gamma\) for every pair of integers \(m\) and \(m'\). Moreover, notice that this relation is commutative; i.e., if \(\alpha_1 \approx_m \alpha_2\), then \(\alpha_2 \approx_m \alpha_1\).

For every pair of consecutive executions in the chain, there is a process that decides on the same value in both executions. By the agreement condition, the decision in \(\alpha_1\) and in \(\alpha_2\) must be the same. This is the main idea of the lower bound proof, which is captured in Theorem 2.2: we take two executions that must have different agreement values and construct an indistinguishability chain between them, which bounds the probability of terminating in terms of the length of the chain. Two such executions exist by the validity condition, as we formalize next.

We partition the processes into \(S = \max\{3, \lceil \frac{n}{f} \rceil \}\) sets \(P_1, \ldots, P_S\), each with at most \(f\) processes. For example, if \(n > 2f\), \(P_i = \{p_{(i-1)f+1}, \ldots, p_{if}\}\) for every \(i, 1 \leq i < S\), and \(P_S = \{p_{(S-1)f+1}, \ldots, p_n\}\).

Consider initial configurations \(C_0, \ldots, C_S\), such that in \(C_0\) all the inputs are 0, and in \(C_i, 1 \leq i \leq S\) all processes in \(P_1, \ldots, P_i\) have input 1 and all other processes have input 0; in particular, in \(C_S\) all processes have input 1.

Let \(\sigma_{\text{full}}\) be the full synchronous schedule with \(k\) layers, in which no process fails. The next theorem is the main tool for bounding \(q_k\) as a function of \(m\), the length of an indistinguishability chain. This theorem distills the technique we borrow from [20]. At the end of section 3 we discuss how asynchrony allows us to construct shorter chains.

**Theorem 2.2.** Assume there is an integer \(m\) such that for any sequences of coins \(\vec{c}\), \(\alpha(\sigma_{\text{full}}, \vec{c}, C_0) \approx_m \alpha(\sigma_{\text{full}}, \vec{c}, C_S)\). Then the probability that \(A\) does not terminate after \(kn - f\) steps is \(q_k \geq \frac{1}{m+1}\).

**Proof.** Assume, by way of contradiction, that \(q_k(m+1) < 1\). Since \(\alpha(\sigma_{\text{full}}, \vec{c}, C_0) \approx_m \alpha(\sigma_{\text{full}}, \vec{c}, C_S)\), there is a chain of \(m+1\) executions,
\[
\alpha(\sigma_{\text{full}}, \vec{c}, C_0) = \beta_1 \overset{p_{\beta_1}}{\sim} \beta_2 \overset{p_{\beta_2}}{\sim} \cdots \overset{p_{\beta_m}}{\sim} \beta_{m+1} = \alpha(\sigma_{\text{full}}, \vec{c}, C_S).
\]

(See Figure 2.1.) The probability that \(A\) does not terminate in at least one of these \(m+1\) executions is at most \(q_k(m+1)\). By assumption, \(q_k(m+1) < 1\), and hence the set \(B\) of sequences of coins \(\vec{c}\) such that \(A\) terminates in all \(m+1\) executions has
probability $\Pr[\bar{c} \in B] > 0$. Since $\alpha(\sigma_{\text{full}}, \bar{c}, C_0) \approx_m \alpha(\sigma_{\text{full}}, \bar{c}, C_S)$, the agreement condition implies that the decision in all $m + 1$ executions is the same. However, the validity condition implies that the decision in $\alpha(\sigma_{\text{full}}, \bar{c}, C_0)$ is 0, and the decision in $\alpha(\sigma_{\text{full}}, \bar{c}, C_S)$ is 1, which is a contradiction. 

A slight extension of the above theorem handles Monte-Carlo algorithms, where processes may terminate without agreement with some small probability $\epsilon$. This extension is presented in section 5.

The statement of Theorem 2.2 indicates that our goal is to show the existence of an integer $m$ such that $\alpha(\sigma_{\text{full}}, \bar{c}, C_0) \approx_m \alpha(\sigma_{\text{full}}, \bar{c}, C_S)$; clearly, the smaller the $m$, the higher the lower bound. The next lemma comes in handy when we construct these chains.

**Lemma 2.3.** Assume there is an integer $m$ such that for every schedule $\sigma$, initial configuration $I$, sequence of coins $\bar{c}$, and set $P_1$, $\alpha(\sigma, \bar{c}, I) \approx_m \alpha(\text{crash}(\sigma, P_1, 1), \bar{c}, I)$. Then $\alpha(\sigma_{\text{full}}, \bar{c}, C_0) \approx_{\sigma(2m+1)} \alpha(\sigma_{\text{full}}, \bar{c}, C_S)$ for all sequences of coins $\bar{c}$.

**Proof.** Consider the schedules $\sigma_0 = \sigma_{\text{full}}$ and $\sigma_i = \text{crash}(\sigma_0, P_1, 1)$ for every $i$, $1 \leq i \leq S$, and the corresponding executions $\alpha_{i,j} = \alpha(\sigma_i, \bar{c}, C_j)$ for every $i$ and $j$, $1 \leq i \leq S$, and $0 \leq j \leq S$. (the execution $\alpha_{i,j}$ starts from the initial configuration $C_j$ with a schedule which is almost full, except that processes in $P_i$ never take any steps).

By assumption, $\alpha_{0,j} \approx_m \alpha_{i,j}$ for every $i$, $1 \leq i \leq S$, and every $j$, $0 \leq j \leq S$. (See Figure 2.2.) Since processes in $P_1$ are crashed in $\sigma_i$ for every $i$, $1 \leq i \leq S$, we have that $\alpha_{i,i-1} \not\approx \alpha_{i,i}$ for every process $p \in P \setminus P_i$. This implies that $\alpha_{i,i-1} \approx_m \alpha_{i,i}$ for every $i$, $1 \leq i \leq S$. Thus,

$$\alpha(\sigma_{\text{full}}, \bar{c}, C_0) = \alpha_{0,0} \approx_m \alpha_{1,0} \approx_m \alpha_{1,1} \approx_m \alpha_{0,1} \approx_m \alpha_{2,1} \approx_m \alpha_{2,2} \cdots \approx_s \alpha_{S,S} \approx_m \alpha_{0,S},$$

where $\alpha_{0,S}$ is exactly $\alpha(\sigma_{\text{full}}, \bar{c}, C_S)$. Therefore, we have $\alpha(\sigma_{\text{full}}, \bar{c}, C_0) \approx_{\sigma(2m+1)} \alpha(\sigma_{\text{full}}, \bar{c}, C_S)$. 

**3. Tradeoff for the message-passing model.** In this section we derive the lower bound for the message-passing model. Notice that in the message-passing model, since a step consists of both sending and receiving messages, a layer $L$ is not only a sequence of processes, but also specifies for each process $p \in L$ the set of processes from which it receives a message (recall that we assumed that it sends messages to all processes). The reception of messages in a certain layer is done after all messages of that layer are sent, and therefore the order of processes in a layer is insignificant.
Formally, an $f$-layer is a sequence $p_{i_1}, \ldots, p_{i_m}$ of distinct process id’s, followed by
a sequence $M_{i_1}, \ldots, M_{i_m}$ of subsets of process id’s, where $M_{i_j}$ is the set of process
id’s from which $p_{i_j}$ receives a message in this layer. In the executions we construct, a
message is either delivered in the same layer, or it is delayed and delivered after the
last layer, and is effectively omitted in the execution.

Recall that the processes are partitioned into $S = \max\{3, \lceil \frac{n}{f} \rceil \}$ sets $P_1, \ldots, P_S$,
each with at most $f$ processes. We manipulate schedules in order to delay messages as follows.

**Definition 3.1.** Let $\sigma$ be a finite schedule. Let $\operatorname{delay}(\sigma, P_i, P_j, r)$ be the schedule
that is the same as $\sigma$, except that the messages sent by processes in $P_i$ in layer $r$ are
received by processes in $P_j$ only after the last layer. More formally, if $M_j$ is the subset
of processes from which a process receives a message in layer $r$ in $\sigma$, then for every
process $p \in P_j$ the subset of processes from which it receives a message in layer $r$ in
$\operatorname{delay}(\sigma, P_i, P_j, r)$ is $M_j \setminus P_i$.

We define indistinguishability of executions in the message-passing model as fol-
lows: two executions are indistinguishable to process $p$ if it goes through the same
local states throughout both executions. More specifically, in both executions $p$ sends
and receives the same messages, in the same order.

Clearly, at the end of layer $r$, any process not in $P_j$ does not distinguish between
the execution so far of a schedule $\sigma$ and an execution so far of $\operatorname{delay}(\sigma, P_i, P_j, r)$. Therefore, we have the following lemma.

**Lemma 3.2.** Let $\sigma$ be a schedule with $k$ layers. For any sequences of coins $\vec{c}$ and
initial configuration $I$, at the end of layer $r$ only processes in $P_j$ distinguish between
$\alpha(\sigma, \vec{c}, I)$ and $\alpha(\operatorname{delay}(\sigma, P_i, P_j, r), \vec{c}, I)$.

Recall that $S = \max\{3, \lceil \frac{n}{f} \rceil \}$ is the number of sets $P_i$. We define the following
recursive function for every $r$ and $k$, $1 \leq r \leq k$:

$$
m_{r,k} = \left\{ \begin{array}{ll}
S & \text{if } r = k, \\
(2(S-1) + 1)m_{r+1,k} + S & \text{if } 1 \leq r < k.
\end{array} \right.
$$

A simple induction shows that $m_{r,k} \leq (2S)^k - r + 1$.

The following lemma proves that $m_{1,k}$ is the integer required in Lemma 2.3 for the
message-passing model, by inductively constructing indistinguishability chains
between executions in which a set of processes may crash from a certain layer $r$.

**Lemma 3.3.** Let $\sigma$ be a schedule with $k$ layers such that for some $r$, $1 \leq r \leq k$, no
process is skipped in layers $r, r+1, \ldots, k$. Then $\alpha(\sigma, \vec{c}, I) \approx m_{r+1,k} \alpha(\operatorname{crash}(\sigma, P_i, r), \vec{c}, I)$
for every sequence of coins $\vec{c}$, every initial configuration $I$, and every $i \in \{1, \ldots, S\}$.

**Proof.** Let $\sigma = \sigma_0$. Throughout this proof we denote $\alpha_i = \alpha(\sigma_i, \vec{c}, I)$ for any
schedule $\sigma_i$. The proof is by backwards induction on $r$.

**Base case $r = k$.** We construct the following schedules. Let $\sigma_1$ be the same as
$\sigma_0$, except that the messages sent by processes in $P_i$ in the $k$th layer are received by
processes in $P_{(i+1) \mod S}$ only after the $k$th layer, i.e., $\sigma_1 = \operatorname{delay}(\sigma, P_i, P_{(i+1) \mod S}, k)$.
By Lemma 3.2, we have $\alpha_0 \preceq \alpha_1$ for every process $p \in P \setminus P_{(i+1) \mod S}$. We continue
inductively to define schedules as above in the following way for every $h$, $0 \leq h \leq S - 1$:

$$
\sigma_{h+1} = \operatorname{delay}(\sigma_h, P_i, P_{(i+h+1) \mod S}, k).
$$

By Lemma 3.2, we have $\alpha_h \preceq \alpha_{h+1}$ for every
process $p \in P \setminus P_{(i+h+1) \mod S}$.

Since in $\sigma_S$ no messages sent by processes in $P_i$ in layer $k$ are ever received, then,
except for local states of the processes in $P_i$, this is the same as if the processes in $P_i$
are crashed in layer $k$:

$$\alpha_S = \alpha(\text{crash}(\sigma, P_i, k), \vec{c}, I),$$

which implies that

$$\alpha(\sigma, \vec{c}, I) \approx_0 \alpha_0 \approx_1 \alpha_1 \approx_1 \cdots \approx_1 \alpha_S = \alpha(\text{crash}(\sigma, P_i, k), \vec{c}, I).$$

Therefore, $\alpha(\sigma, \vec{c}, I) \approx_S \alpha(\text{crash}(\sigma, P_i, k), \vec{c}, I)$.

**Induction step.** Informally, this is similar to the base case, except that we crash $P_j$ in layer $r + 1$ before “erasing” messages from $P_i$ to $P_j$ in layer $r$, and afterwards revive $P_j$ in layer $r + 1$.

Formally, we assume that the lemma holds for layer $r + 1$, $1 \leq r < k$, and prove that it holds for layer $r$. Let $\sigma_1 = \text{crash}(\sigma_0, P_{(i+1)} \mod S, r + 1)$; by the induction hypothesis, $\alpha_0 \approx_{m_{r+1,k}} \alpha_1$.

Let $\sigma_2$ be the same as $\sigma_1$, except that the messages received by processes in $P_{(i+1)} \mod S$ from processes in $P_i$ in layer $r$ are received only after the $k$th layer, i.e., $\sigma_2 = \text{delay}(\sigma_1, P_i, P_{(i+1)} \mod S, r)$. By Lemma 3.2, at the end of layer $r$ only processes in $P_{(i+1)} \mod S$ distinguish between the executions, but since they are crashed in layer $r + 1$ we have $\alpha_1 \approx \alpha_2$, for every process $p \in P \setminus P_{(i+1)} \mod S$, implying that $\alpha_1 \approx \alpha_2$.

Let $\sigma_3$ be the same as $\sigma_2$, except that the processes in $P_{(i+1)} \mod S$ do not crash in layer $r + 1$. This implies that

$$\sigma_2 = \text{crash}(\sigma_3, P_{(i+1)} \mod S, r + 1).$$

By the induction hypothesis, we have $\alpha_2 \approx_{m_{r+1,k}} \alpha_3$. (See Figure 3.1.)

We continue inductively to define schedules as above in the following way for every $h$, $0 \leq h \leq S - 1$. We define $\sigma_{3h+1} = \text{crash}(\sigma_{3h}, P_{(i+h+1)} \mod S, r + 1)$, and therefore by the induction hypothesis $\sigma_{3h} \approx_{m_{r+1,k}} \alpha_{3h+1}$. Let $\alpha_{3h+2}$ be the same as $\alpha_{3h+1}$, except that the messages received by processes in $P_{(i+h+1)} \mod S$ from processes in $P_i$ in layer $r$ are received only after the $k$th layer, i.e., $\sigma_{3h+2} = \text{delay}(\sigma_{3h+1}, P_i, P_{(i+h+1)} \mod S, r)$. By Lemma 3.2, at the end of layer $r$ only processes in $P_{(i+h+1)} \mod S$ distinguish between the executions, but since they are crashed in layer $r + 1$ we have $\alpha_{3h+1} \approx \alpha_{3h+2}$, for every process $p \in P \setminus P_{(i+h+1)} \mod S$, implying that $\alpha_{3h+1} \approx_1 \alpha_{3h+2}$.

Finally, we define $\alpha_{3h+3}$ to be the same as $\alpha_{3h+2}$, except that processes in $P_{(i+h+1)} \mod S$ do not crash. This implies that $\sigma_{3h+2} = \text{crash}(\sigma_{3h+3}, P_{(i+h+1)} \mod S, r + 1)$. By the induction hypothesis we have $\alpha_{3h+2} \approx_{m_{r+1,k}} \alpha_{3h+3}$.
The construction implies that in $\sigma_{3(S-1)+2}$ no messages are sent by the processes in $P_i$, in layer $r$, and they are crashed from layer $r+1$. Except for local states of the processes in $P_i$, this is the same as if the processes in $P_i$ are crashed from layer $r$. Therefore

$$\alpha(\sigma_{3(S-1)+2}, \vec{c}, I) = \alpha(\text{crash}(\sigma_0, P_i, r), \vec{c}, I),$$

and hence

$$\alpha_0 \approx m_{r+1,k} \alpha_1 \approx 1 \alpha_2 \approx m_{r+1,k} \alpha_3 \approx m_{r+1,k} \cdots \approx m_{r+1,k} \alpha_3(S-1)+1 \approx 1 \alpha_3(S-1)+2.$$ 

Since $m_{r,k} = (2(S-1)+1)m_{r+1,k}+S$, we have $\alpha_0 \approx m_{r,k} \alpha(\text{crash}(\sigma_0, P_i, r), \vec{c}, I)$. 

Note that in all executions constructed in the proof, at most one set of processes $P_i$ does not appear in a layer; since $|P_i| \leq f$, this implies that at least $n-f$ processes take a step in every layer, and hence every execution in the construction contains at least $k(n-f)$ steps.

Lemmas 2.3 and 3.3 imply that for any sequence of coins $\vec{c}$, $\alpha(\sigma_{\text{full}}, \vec{c}, C_0) \approx S(2m_{1,k}+1) \alpha(\sigma_{\text{full}}, \vec{c}, C_S)$. Since $m_{1,k} \leq (2S)k$, substituting $S(2m_{1,k}+1)$ in the parameter $m$ of Theorem 2.2 yields that $q_k \geq \frac{1}{(2S)^{k+1}}$. Recall that $S = \max\{3, \lceil \frac{f}{7} \rceil \}$. Taking $\lceil \frac{f}{7} \rceil$ to be a constant, we obtain the main result of this section.

**Theorem 3.4.** Let $A$ be a randomized consensus algorithm in the asynchronous message-passing model. There is a weak adversary and an initial configuration, such that the probability that $A$ does not terminate after $k(n-f)$ steps is at least $\frac{1}{c}$, where $c$ is a constant if $\lceil \frac{f}{7} \rceil$ is a constant.

In the original construction for the synchronous model ([21, 22]; see also [6, Chapter 5]), a process that does not appear in a round $r$ must be crashed in that round, and therefore must be counted within the $f$ failures allowed. Hence, in order to change all the inputs from 0 to 1, we must crash and revive fewer processes at a time at each round. For example, in order to continue $k \leq f$ rounds only one process may be crashed at each round. This adds a factor of $k$ to the base of the power in the denominator of the bound on $q_k$, which results in a lower bound of $\frac{1}{ck^k}$ for the synchronous message-passing model [20].

### 4. Trade-off for the shared-memory model

We now derive a similar lower bound for two shared-memory models, where processes communicate through shared read/write registers. The first model consists of single-writer registers and a snapshot operation that costs one step, described formally in subsection 4.1. In subsection 4.2 we consider multiwriter registers. The lower bounds clearly hold for the more restricted model, where processes read only a single register in each memory access.

In the shared-memory model, the definition of indistinguishability is slightly different than in the message-passing model. For two executions $\alpha$ and $\alpha'$ to be indistinguishable to a process $p$, we do not only require $p$ to have the same local states throughout both executions, but also that the values of the shared registers are the same throughout both; otherwise, for example, having $p$ perform a read operation after $\alpha$ and $\alpha'$ might result in different executions. This implies that in both executions $p$ performs the same shared-memory operations, including reading the same values from registers. However, we allow the value of a shared register to be different at the end of $\alpha$ and $\alpha'$ if it is no longer accessed by any process. This slight modification still captures the requirement that if a process $p$ decides $v$ in $\alpha$ and does not distinguish between $\alpha$ and $\alpha'$, then it also decides in $\alpha'$ on the same value $v$. 

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.
4.1. Single-writer cheap-snapshot. We first consider a shared-memory model where processes communicate through single-writer registers. The lower bound is proved under a simplifying assumption that each read step accesses the registers of all processes. We call this the single-writer cheap-snapshot model, since each register is written to by one specific process, and all registers are read by any process in a single snapshot. This snapshot is charged one step, hence the term “cheap.”

As in a standard shared-memory model, a step of a process consists of accessing the shared memory and performing local computations. We further assume that in the algorithm the steps of every process alternate between a write and a cheap-snapshot, starting with a write. Any algorithm can be transformed to satisfy this requirement by having a process rewrite the same value to its register if it is forced to take a write operation, or read all of the registers and ignore some of (or all) their values if it is forced to take a cheap-snapshot operation. This only doubles the step complexity.

Recall that the processes are partitioned into \( S = \max\{3, \lceil \frac{n}{f} \rceil \} \) sets \( P_1, \ldots, P_S \), each with at most \( f \) processes. We consider a restricted set of layered schedules.

**Definition 4.1.** A schedule \( \sigma \) is regular if for every layer \( L \) and every \( i \), \( 1 \leq i \leq S \), either all processes \( p \in P_i \) take a step in \( L \) consecutively (one after the other, without steps of processes not in \( P_i \) in between), or none of the processes \( p \in P_i \) takes a step in \( L \). We denote by \( \pi \) the permutation of the sets \( P_i \) that take steps in \( L \); i.e., if processes \( p \in P_i \) take a step in \( L \), then \( \pi^{-1}(i) \) is their index in the layer. We denote by \( |\pi| \) the number of sets \( P_i \) that take steps in the layer.

Note that, in contrast to the message-passing model, in a shared-memory model the order of the processes in a layer \( L \) is significant, since different orderings result in different executions.

Regular schedules are useful in our proofs since in every layer all the processes in some set \( P_i \) perform the same operation, as argued in the next lemma. Since processes in the same set \( P_i \) either all write to different registers (recall that registers are single-writer) or read all registers, this means that in a regular execution the order of processes in the set \( P_i \) does not matter.

**Lemma 4.2.** Let \( \sigma \) be a regular schedule with \( k \) layers. Then in every layer \( L \) in \( \sigma \), for every \( i \), \( 1 \leq i \leq S \), either all process \( p \in P_i \) do not take a step in \( L \), or all processes \( p \in P_i \) perform a write operation in \( L \), or all processes \( p \in P_i \) perform a cheap-snapshot operation in \( L \).

*Proof.* The proof is by induction on the layer number \( r \).

**Base case.** Let \( r = 1 \); i.e., \( L \) is the first layer of \( \sigma \). Since \( \sigma \) is regular, either all process \( p \in P_i \) take a step in \( L \), or none of the processes \( p \in P_i \) take a step in \( L \). If all take a step, then by our assumption on the algorithm, it is a write operation. Otherwise, none takes a step, which proves the base case.

**Induction step.** Assume the lemma holds for layer \( \ell \), \( 1 \leq \ell \leq r \). We prove the lemma for layer \( r + 1 \). By the induction hypothesis, in every layer \( \ell \), \( 1 \leq \ell \leq r \), either all processes \( p \in P_i \) perform a cheap-snapshot operation, or all perform a write operation, or none performs an operation. If none performs any operation in any layer \( \ell \leq r \), then at the beginning of layer \( r + 1 \) the pending operation of all processes \( p \in P_i \) is a write operation by our assumption on the algorithm. Otherwise, let \( \ell \) be the maximal layer in which all processes \( p \in P_i \) took a step. If they are cheap-snapshot operations, then at the beginning of layer \( r + 1 \) the pending operation of all processes \( p \in P_i \) is a cheap-snapshot operation by our assumption on the algorithm. If they are write operations, then at the beginning of layer \( r + 1 \) the pending operation of all processes \( p \in P_i \) is a write operation by our assumption on the algorithm. In any case, at the beginning of layer \( r + 1 \), either all processes \( p \in P_i \) have a pending
cheap-snapshot operation, or all have a pending write operation. Since \( \sigma \) is regular, either none of the processes \( p \in P_i \) takes a step in layer \( r+1 \), or all take a step in layer \( r+1 \), in which case it would either be a cheap-snapshot operation for all processes, or a write operation for all processes.

In the proof, we apply certain manipulations to regular schedules, allowing us to delay and crash sets of processes as follows.

**Definition 4.3.** Let \( \sigma \) be a schedule such that every \( p \in P_i \) is nonfaulty in layer \( r \), and such that \( P_i \) is not the last set of processes in the layer. Let \( \text{swap}(\sigma, P_i, r) \) be the schedule that is the same as \( \sigma \), except that the steps of processes in \( P_i \) are swapped with steps of the next set of processes in that layer. Formally, if \( \pi \) is the permutation of layer \( r \) in \( \sigma \) and \( \pi' \) is the permutation of layer \( r \) in \( \text{swap}(\sigma, P_i, r) \), and if \( j = \pi^{-1}(i) \), then we have \( \pi'(j) = \pi(j+1) \) and \( \pi'(j+1) = \pi(j) \).

Inductively, we define

\[
\text{swap}^i(\sigma, P_i, r) = \text{swap}(\text{swap}^{i-1}(\sigma, P_i, r), P_i, r);
\]

that is, \( P_i \) is swapped \( j \) times and moved \( j \) sets later in the layer.

**Definition 4.4.** Let \( \sigma \) be a schedule and \( r \) be a layer such that no process is skipped in any layer \( \ell > r \). Let \( \text{delay}(\sigma, P_i, r) \) be the schedule that is the same as \( \sigma \), except that the steps of \( P_i \) starting from layer \( r \) are delayed by one layer. Thus, there is no step of \( p \in P_i \) in layer \( r \), the step of \( p \in P_i \) in layer \( r+1 \) is the step that was in layer \( r \), and so on. The permutations of the layers \( \ell \geq r+1 \) do not change.

Note that this definition assumes a schedule \( \sigma \) in which no process is skipped in any layer \( \ell > r \). Specifically, this implies that \( P_i \) appears in every layer \( \ell \geq r+1 \), which allows us to keep the permutations in layers \( \ell \geq r+1 \) unchanged in \( \text{delay}(\sigma, P_i, r) \).

Delaying a set \( P_i \) from layer \( r \) can be seen as delaying \( P_i \) from layer \( r+1 \), swapping \( P_i \) in layer \( r \) until it reaches the end of the layer, accounting for \( P_i \) as the first set in layer \( r+1 \) instead of the last set in layer \( r \), and then swapping \( P_i \) in layer \( r+1 \) until it reaches its original place in the layer.

Although accounting for \( P_i \) as the first set in layer \( r+1 \) instead of the last set in layer \( r \) does not change the order of steps taken, it is technically a different schedule (recall that the schedules are defined as sequences of layers, which in this case are different in layers \( r \) and \( r+1 \)). Therefore we define the following definition.

**Definition 4.5.** Let \( \sigma \) be a schedule where the last set of processes in layer \( r \) is \( P_i \), and this set does not appear in layer \( r+1 \). Let \( \text{rollover}(\sigma, P_i, r) \) be the schedule that is the same as \( \sigma \), except that \( P_i \) is the first set in layer \( r+1 \) instead of the last set in layer \( r \).

Effectively, such two schedules \( \sigma \) and \( \text{rollover}(\sigma, P_i, r) \) have the same order of steps, which implies that the executions of these schedules is the same:

\[
\alpha(\sigma, \bar{c}, I) = \alpha(\text{rollover}(\sigma, P_i, r), \bar{c}, I).
\]

Definitions 4.3, 4.4, and 4.5 imply the following corollary.

**Corollary 4.6.** Let \( \sigma \) be a regular schedule with \( k \) layers. For every \( r, 1 \leq r \leq k \), and \( \pi_r \) the permutation of layer \( r \) in \( \sigma \),

\[
\text{delay}(\sigma, P_i, r) = \text{swap}^{\pi_r^{-1}(i)-1}(\text{rollover}(\text{swap}^{\pi_r-\pi_r^{-1}(i)}(\text{delay}(\sigma, P_i, r+1), P_i, r), P_i, r), P_i, r+1).
\]

Figure 4.1 depicts the schedules used when delaying a set \( P_i \) in layer \( r \) of a schedule \( \sigma \), according to this corollary.
Recall that crash(σ, Pᵢ, r) is the schedule that is the same as σ, except that processes in Pᵢ crash in layer r. Crashing a set Pᵢ in layer r can be seen as delaying it from layer r and then crashing it from layer r + 1. Definitions 2.1 and 4.4 imply the following corollary.

**Corollary 4.7.** For every regular schedule σ,

\[
\text{crash}(\sigma, Pᵢ, r) = \text{crash}(\text{delay}(\sigma, Pᵢ, r), Pᵢ, r + 1).
\]

An important property of regular schedules is that swapping, delaying, or crashing a set of processes Pᵢ yields a regular schedule as well, because the sets are manipulated together.

**Lemma 4.8.** Let σ be a regular schedule with k layers. Then for every i, 1 ≤ i ≤ S, and every r, 1 ≤ r ≤ k, the schedules swap(σ, Pᵢ, r), delay(σ, Pᵢ, r), rollover(σ, Pᵢ, r), and crash(σ, Pᵢ, r) are regular.

**Proof.** Every layer ℓ ≠ r in swap(σ, Pᵢ, r) is the same as in σ and therefore satisfies the requirement of a regular schedule. In layer r, all processes that took steps in σ also take steps in swap(σ, Pᵢ, r), and each set remains consecutive. Therefore, swap(σ, Pᵢ, r) is regular. It is also easy to see that rollover(σ, Pᵢ, r) is regular.

The proof for delay(σ, Pᵢ, r) and crash(σ, Pᵢ, r) is by backwards induction on the layer number r.

**Base case.** For r = k, delaying a set Pᵢ in the last layer is the same as crashing Pᵢ. Denote σ' = delay(σ, Pᵢ, k) = crash(σ, Pᵢ, k). Every layer ℓ < k in σ' is the same as in σ, and the last layer k is the same in σ', except that the processes in Pᵢ do not take a step. Hence, σ' is also regular.

**Induction step.** We assume the statement holds for layer ℓ, r + 1 ≤ ℓ ≤ k, and prove it for layer r. By Corollary 4.6 and the induction hypothesis, and since swapping results in a regular schedule, delay(σ, Pᵢ, r) is regular. By Corollary 4.7 and the

\[
\text{delay}(\sigma, Pᵢ, r) = \text{delay}(\text{delay}(\sigma, Pᵢ, r), Pᵢ, r + 1).
\]

Fig. 4.1. An example showing how swap operators are applied to delay a set of processes Pᵢ; assume Pᵢ is the penultimate set in layer r and the third set in layer r + 1. Note that the third transition does not modify the execution and only accounts for the steps of Pᵢ to layer r + 1 instead of layer r; the last transition just notes that we have obtained delay(σ, Pᵢ, r).
induction hypothesis, and since delaying results in a regular schedule, \(\text{crash}(\sigma, P_i, r)\) is regular.  

We next construct an indistinguishability chain of schedules between any regular schedule and a schedule in which some set of processes is delayed or crashed. The construction relies on Corollaries 4.6 and 4.7 to delay or crash a set of processes through a sequence of swaps. The elementary step in this construction, where a set is swapped with the following one, is provided by the next lemma.

**Lemma 4.9.** Let \(\sigma\) be a regular schedule with \(k\) layers. For any sequences of coins \(\vec{c}\) and initial configuration \(I\), if \(P_i\) is not the last set of processes in layer \(r\), \(1 \leq r \leq k\), then there is a set \(P_j\) such that at the end of layer \(r\) only processes in \(P_j\) (at most) distinguish between \(\alpha(\sigma, \vec{c}, I)\) and \(\alpha(\text{swap}(\sigma, P_i, r), \vec{c}, I)\).

**Proof.** Consider \(\text{swap}(\sigma, P_i, r)\) and let \(\pi\) be the permutation corresponding to layer \(r\). Since \(P_i\) is not the last set in the layer, we have \(\pi^{-1}(i) \neq |\pi|\). Let \(i' = \pi(\pi^{-1}(i)+1)\); i.e., \(P_i\) is swapped with \(P_{i'}\). By Lemma 4.2, either all the processes in \(P_i\) perform a cheap-snapshot operation or all processes in \(P_{i'}\) perform a write operation. The same applies for \(P_{i'}\).

If both types of operations are cheap-snapshot operations or both types are write operations (necessarily to different registers), then \(\alpha(\sigma, \vec{c}, I) \sim \alpha(\text{swap}(\sigma, P_i, r), \vec{c}, I)\) for every process \(p\) in the layer.

If one type of operation is cheap-snapshot and the other is writing, then only the processes in the set performing cheap-snapshot observe a difference. Denote this set by \(P_j\) (where \(j\) is either \(i\) or \(i'\)). Notice that the set \(P_j\) (the value of the index \(j\)) depends only on the types of operations performed, i.e., only on \(\sigma\), and not on the sequences of coins \(\vec{c}\) or the initial configuration \(I\). This is necessary for claiming that the adversary is nonadaptive.

For every \(r\) and \(k\), \(1 \leq r \leq k\), we define

\[
\begin{align*}
    s_{r,k} &= \begin{cases} 
        1 & \text{if } r = k, \\
        2 \cdot c_{r+1,k} + 1 & \text{if } 1 \leq r < k,
    \end{cases} \\
    d_{r,k} &= \begin{cases} 
        S & \text{if } r = k, \\
        d_{r+1,k} + S \cdot s_{r,k} + S \cdot s_{r+1,k} & \text{if } 1 \leq r < k,
    \end{cases} \\
    c_{r,k} &= \begin{cases} 
        S & \text{if } r = k, \\
        d_{r,k} + c_{r+1,k} & \text{if } 1 \leq r < k,
    \end{cases}
\end{align*}
\]

where \(S = \max\{3, \lceil \frac{1}{2} \rceil \}\) is the number of sets \(P_i\). These recursive functions will be used for bounding the lengths of the indistinguishability chains in our construction.

The next proposition shows a bound on these functions; its proof appears in [14].

**Proposition 4.10.** \(c_{r,k} \leq (2S + 4)^{r-1}\).

The main technical result of this section is the following lemma, which will be used to show an indistinguishability chain between the executions that result from schedules \(\sigma\) and \(\text{crash}(\sigma, P_i, 1)\) in order to apply Lemma 2.3. Additional claims, regarding \(\text{swap}\) and \(\text{delay}\), are proved in order to carry through with the proof.

**Lemma 4.11.** Let \(\sigma\) be a regular schedule with \(k\) layers such that no process is skipped at any layer \(l \geq r\) for some \(r\), \(1 \leq r \leq k\). For any sequences of coins \(\vec{c}\) and initial configuration \(I\), and for every \(i\), \(1 \leq i \leq S\), the following hold:

\[
\begin{align*}
    \alpha(\sigma, \vec{c}, I) &\approx_{s_{r,k}} \alpha(\text{swap}(\sigma, P_i, r), \vec{c}, I), \\
    \alpha(\sigma, \vec{c}, I) &\approx_{d_{r,k}} \alpha(\text{delay}(\sigma, P_i, r), \vec{c}, I), \\
    \alpha(\sigma, \vec{c}, I) &\approx_{c_{r,k}} \alpha(\text{crash}(\sigma, P_i, r), \vec{c}, I).
\end{align*}
\]
Proof. Let \( \sigma_0 = \sigma \). Throughout the proof, we denote \( \alpha_i = \alpha(\sigma_i, \vec{c}, I) \) for every schedule \( \sigma_i \), and \( \alpha'_i = \alpha(\sigma'_i, \vec{c}, I) \) for every schedule \( \sigma'_i \).

The proof is by backwards induction on \( r \).

Base case \( r = k \). Consider swap(\( \sigma, P_i, k \)), where \( P_i \) is not the last set in the layer (otherwise swapping is undefined). By Lemma 4.9, there is a set \( P_j \), which does not depend on \( \vec{c} \) or \( I \), such that \( \alpha(\sigma, \vec{c}, I) \not\approx \alpha(\text{swap}(\sigma, P_i, r), \vec{c}, I) \) for every process \( p \not\in P_j \). Therefore, \( \alpha(\sigma, \vec{c}, I) \approx_{s_{k,k}} \alpha(\text{swap}(\sigma, P_i, k), \vec{c}, I) \).

Delaying \( P_i \) in the last layer is equivalent to failing it; therefore delay(\( \sigma, P_i, k \)) = crash(\( \sigma, P_i, k \)). Denote this schedule by \( \sigma'' \). We crash \( P_i \) by swapping it until it reaches the end of the layer and then removing it. In more detail, let \( \pi \) be the permutation of the last layer of \( \sigma \), and define

\[
\sigma'' = \text{swap}^{\pi | - \pi^{-1}(i)}(\sigma, P_i, k).
\]

The proof of the base case for swap(\( \sigma, P_i, k \)) implies that there is a chain of length \( s_{k,k} \) \cdot (|\pi_r| - \pi_r^{-1}(i)) \leq (S - 1) \cdot s_{k,k} = S - 1 \) between the executions, i.e., \( \alpha_0 \approx_{S-1} \alpha(\sigma'', \vec{c}, I) \). Clearly, \( \alpha(\sigma'', \vec{c}, I) \not\approx \alpha(\sigma', \vec{c}, I) \) for every process \( p \not\in P_i \), and therefore \( \alpha(\sigma, \vec{c}, I) \approx_{\sigma'_i, \vec{c}, I} \alpha(\text{delay}(\sigma, P_i, r), \vec{c}, I) \) and \( \alpha(\sigma, \vec{c}, I) \approx_{\sigma'_i, \vec{c}, I} \alpha(\text{crash}(\sigma, P_i, r), \vec{c}, I) \).

Induction step. Assume the statement holds for layer \( r + 1 \leq k \). We prove that it holds for layer \( r \).

We first deal with swapping. Assume that \( P_i \) is not the last set in the layer and consider swap(\( \sigma, P_i, r \)). By Lemma 4.9, there is a set \( P_j \), which does not depend on \( \vec{c} \) or \( I \), such that at the end of layer \( r \) only processes in \( P_j \) distinguish between \( \alpha(\sigma, \vec{c}, I) \) and \( \alpha(\text{swap}(\sigma, P_i, r), \vec{c}, I) \). We define \( \sigma_1 \) to be the same as \( \sigma \), except that processes in \( P_j \) are crashed in layer \( r + 1 \), i.e., \( \sigma_1 = \text{crash}(\sigma, P_j, r + 1) \). By the induction hypothesis, \( \alpha_0 \approx_{\sigma_1, \vec{c}, I} \alpha_1 \). Let \( \sigma_2 \) be the same as \( \sigma_1 \), except that \( P_i \) and \( P_j \) are swapped in layer \( r \), i.e., \( \sigma_2 = \text{swap}(\sigma_1, P_i, r) \). Since only processes in \( P_j \) observe the swapping, but are all crashed in the next layer, we have that \( \alpha_1 \not\approx \alpha_2 \) for every process \( p \not\in P_j \). Finally, let \( \sigma_3 \) be the same as \( \sigma_2 \), except that processes in \( P_i \) are not crashed in layer \( r + 1 \), i.e., \( \sigma_2 = \text{crash}(\sigma_3, P_j, r + 1) \). By the induction hypothesis, \( \alpha_2 \approx_{\sigma_3, \vec{c}, I} \alpha_3 \). Notice that \( \sigma_3 = \text{swap}(\sigma, P_i, r) \), and \( 2s_{r+1,k} + 1 = s_{r,k} \), which implies that \( \alpha(\sigma, \vec{c}, I) \approx_{s_{r,k}} \alpha(\text{swap}(\sigma, P_i, r), \vec{c}, I) \).

Next, we consider the case of delaying a process, i.e., delay(\( \sigma, P_i, r \)). (Recall Figure 4.1.) By Corollary 4.6,

\[
delay(\sigma, P_i, r) = \text{swap}^{\pi_r^{-1}(i)-1}(\text{rollover}(\text{swap}^{\pi_r| - \pi_r^{-1}(i)}(\text{delay}(\sigma, P_i, r + 1), P_i, r), P_i, r), P_i, r + 1).
\]

Recall that applying rollover does not change the execution. Hence, by the proof for swapping and the induction hypothesis, and since

\[
d_{r+1,k} + s_{r,k} \cdot (|\pi_r| - \pi_r^{-1}(i)) + s_{r+1,k} \cdot (\pi_r^{-1}(i) - 1) \leq d_{r+1,k} + S \cdot s_{r,k} + S \cdot s_{r+1,k} = d_{r,k},
\]

it follows that \( \alpha(\sigma, \vec{c}, I) \approx_{d_{r,k}} \alpha(\text{delay}(\sigma, P_i, r), \vec{c}, I) \).

Finally, we consider the case of crashing a process, i.e., crash(\( \sigma, P_i, r \)). By Corollary 4.7,

\[
\text{crash}(\sigma, P_i, r) = \text{crash}(\text{delay}(\sigma, P_i, r), P_i, r + 1).
\]

By the proof for delaying and the induction hypothesis, and since \( d_{r,k} + c_{r+1,k} = c_{r,k} \), it follows that

\[
\alpha(\sigma, \vec{c}, I) \approx_{c_{r,k}} \alpha(\text{crash}(\sigma, P_i, r), \vec{c}, I).
\]
Algorithm 1. Simulating a multiwriter register $R$ from single-writer registers

Process $p_i$ has a shared register $RR[1]$, each consisting of the pair $⟨v,t⟩$; initially, each register holds $⟨0,init⟩$, where $init$ is the desired initial value

1: $read(R)$:
2: snapshot $RR$ array into local $(t,v)$ array
3: return $v[j]$ such that $t[j]$ is maximal

4: $write(R,v)$ by $p_w$:
5: snapshot $RR$ array into local $(t,v)$ array
6: let $lts$ be the maximum of $t[1],...,t[n]$
7: write the pair $⟨v,lts+1⟩$ to $RR[w]$
8: return

Note that in all executions constructed in the proof, at most one set of processes $P_i$ does not appear in a layer; since $|P_i| ≤ f$, this implies that at least $n − f$ processes take a step in every layer, and hence every execution in the construction contains at least $k(n − f)$ steps.

Lemmas 2.3 and 4.11 imply that for every sequence of coins $c$, $α(σ_{full}, c, C_0) ≈ S(2c_1,k+1)+1 \cdot α(σ_{full}, c, C_0)$, since $c_1,k ≤ (2S + 4)^k$, we have that $S(2c_1,k+1)+1 ≤ (2S + 4)^k+1$. Substituting in Theorem 2.2 yields that $q_k ≥ \frac{1}{S(2S+4)^k+1}$. Since $S$ can be taken to be a constant when $\lceil \frac{n}{f} \rceil$ is a constant, we get the following theorem.

Theorem 4.12. Let $A$ be a randomized consensus algorithm in the asynchronous shared-memory model, with single-writer registers and cheap-snapshots. There is a weak adversary and an initial configuration, such that the probability that $A$ does not terminate after $k(n − f)$ steps is at least $\frac{1}{c}$, where $c$ is a constant if $\lceil \frac{n}{f} \rceil$ is a constant.

4.2. Multiwriter cheap-snapshot. When multiwriter registers are considered, the above approach cannot be directly used. This is because swapping two sets of processes which perform write operations may change the value of a register and hence may lead other processes, which later read (or perform a cheap-snapshot), to distinguish this from before the swap. This cannot happen when registers are single-writer, since no two processes write to the same register, and therefore swapping two write operations is indistinguishable to all processes.

Instead, we derive the lower bound for multiwriter registers by reduction to single-writer registers. In a simple simulation of a multiwriter register from single-writer registers (e.g., [33]), performing a high-level read or write operation (to the multiwriter register) involves $n$ low-level read operations (of all single-writer registers) and possibly one low-level write operation (to the process’ own single-writer register). This multiplies the total step complexity by $O(n)$.

However, with cheap-snapshots, we can read all single-writer registers in one step, yielding a simulation that only doubles the total step complexity (since writing includes a cheap-snapshot operation). The pseudocode of the simulation appears in Algorithm 1, which uses an array $RR$ of single-writer variables. $RR[i]$ is the last value written by $p_i$, together with a timestamp. The correctness of this algorithm follows along the lines of the proof of Algorithm 10.3 from [6].

Since in the single-writer cheap-snapshot model each snapshot operation accounts for one access to the shared memory, every algorithm in the multiwriter model can be...
transformed to an algorithm in the single-writer cheap-snapshot model, where the step complexity is multiplied only by a constant factor. Combining this with Theorem 4.12 yields the following theorem.

**Theorem 4.13.** Let $A$ be a randomized consensus algorithm in the asynchronous shared-memory model, with multiwriter registers and cheap-snapshots. There is a weak adversary and an initial configuration, such that the probability that $A$ does not terminate after $k(n - f)$ steps is at least $\frac{1}{m}$, where $c$ is a constant if $\lceil \frac{n}{k} \rceil$ is a constant.

5. Monte-Carlo algorithms. Another way to overcome the impossibility of asynchronous consensus is to allow Monte-Carlo algorithms. This requires us to relax the agreement property and allow the algorithm to decide on conflicting values, with small probability. Let $\epsilon_k$ be the maximum probability, over all weak adversaries and over all initial configurations, that processes decide on conflicting values after $k(n - f)$ steps. The next theorem is the analogue of Theorem 2.2, for bounding the probability of terminating after $k(n - f)$ steps.

**Theorem 5.1.** Assume there is an integer $m$ such that for any sequences of coins $\vec{c}$, $\alpha_0(\sigma_{full}, \vec{c}, C_0) \approx_m \alpha(\sigma_{full}, \vec{c}, C_S)$. Then $q_k \geq \frac{1-(m+1)\epsilon_k}{m+1}$.

Proof. Assume, by way of contradiction, that $(m+1)q_k < 1 - (m+1)\epsilon_k$. Consider the $m+1$ executions in the sequence implied by the fact that $\alpha(\sigma_{full}, \vec{c}, C_0) \approx_m \alpha(\sigma_{full}, \vec{c}, C_S)$. The probability that $A$ does not terminate in at least one of these $m+1$ executions is at most $(m+1)q_k$. By assumption, $q_k(m+1) < 1 - (m+1)\epsilon_k$, and hence the set $B$ of sequences of coins $\vec{c}$ such that $A$ terminates in all $m+1$ executions has probability $Pr[\vec{c} \in B] > (m+1)\epsilon_k$.

If the agreement property is satisfied in all $m+1$ executions, then by the validity condition, as in the proof of Theorem 2.2, we get that the decision in $\alpha(\sigma_{full}, \vec{c}, C_0)$ is the same as the decision in $\alpha(\sigma_{full}, \vec{c}, C_S)$, which is a contradiction. Hence, for every $\vec{c} \in B$, the agreement condition does not hold in at least one of these executions.

Since we have $m+1$ schedules in the chain, there exists a schedule for which the agreement condition does not hold with probability greater than $\epsilon_k$. But this means that $A$ satisfies agreement with probability smaller than $1 - \epsilon_k$, which is a contradiction. $\blacksquare$

Substituting with Lemmas 2.3 and 3.3 yields the lower bound for the message-passing model.

**Theorem 5.2.** Let $A$ be a randomized consensus algorithm in the asynchronous message-passing model. There is a weak adversary and an initial configuration, such that the probability that $A$ does not terminate after $k(n - f)$ steps is at least $\frac{1}{m}$, where $c$ is a constant if $\lceil \frac{n}{k} \rceil$ is a constant, and $\epsilon_k$ is a bound on the probability for disagreement.

Substituting with Lemma 2.3 and the proof of Theorem 4.13 yields the lower bound for the shared-memory model.

**Theorem 5.3.** Let $A$ be a randomized consensus algorithm in the asynchronous shared-memory model with multiwriter registers and cheap-snapshots. There is a weak adversary and an initial configuration, such that the probability that $A$ does not terminate after $k(n - f)$ steps is at least $\frac{1}{m}$, where $c$ is a constant if $\lceil \frac{n}{k} \rceil$ is a constant, and $\epsilon_k$ is a bound on the probability for disagreement.

The bound we obtain in Theorem 5.1 on the probability $q_k$ of not terminating increases as the allowed probability $\epsilon_k$ of terminating without agreement decreases, and coincides with Theorem 2.2 in case the agreement property must always be satisfied (i.e., $\epsilon_k = 0$). In case an algorithm always terminates in $k(n - f)$ steps (i.e., $q_k = 0$), we can restate Theorem 5.1 as a bound on $\epsilon_k$. 

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.
Corollary 5.4. Assume there is an integer \( m \) such that for any sequences of coins \( c, \sigma ( f, c, C_0, C_S) \approx \sigma ( f, c, C_0, C_S) \). Moreover, assume that the algorithm always terminates after \( k(n - f) \) steps, i.e., \( q_k = 0 \). Then \( \epsilon_k \geq \frac{1}{m+1} \).

For example, the algorithms for the message-passing model given by Kapron et al. [27] are Monte-Carlo; i.e., they have a small probability for terminating without agreement. They present an algorithm that always terminates within \( \text{polylog}(n) \) asynchronous rounds and has a probability \( \frac{1}{\text{polylog}(n)} \) for disagreeing. For comparison, our lower bound of Corollary 5.4 for disagreeing when \( k = \text{polylog}(n) \) and \( q_k = 0 \) is \( \epsilon_k \geq \frac{1}{\text{polylog}(n)} \), where \( c \) is a constant if \( \lceil \frac{n}{f} \rceil \) is a constant. Their second algorithm always terminates within \( 2^{\Theta(\log^7 n)} \) asynchronous rounds and has a probability \( \frac{1}{\text{poly}(n)} \) for disagreeing, while our lower bound is \( \epsilon_k \geq \frac{1}{c^2 \text{poly}(n)} \).

6. Discussion. We presented lower bounds for the termination probability achievable by randomized consensus algorithms with bounded running time, under a very weak adversary. Our results are particularly relevant in light of the recent surge of interest in providing Byzantine fault-tolerance in practical, asynchronous distributed systems (e.g., [13, 30]). The adversarial behavior in these applications is better captured by nonadaptive adversaries as used in our lower bounds, rather than the adaptive adversary, which can observe the results of local coin-flips used in most previous lower bounds [1, 5, 10].

For all models, when agreement is required to always hold, we have shown that the probability \( q_k \) that the algorithm fails to complete in \( k(n - f) \) steps is at least \( \frac{1}{c^k} \) for a model-dependent value \( c \) which is a constant if \( \lceil \frac{n}{f} \rceil \) is a constant. Table 6.1 shows the bounds for specific values of \( k \).

<table>
<thead>
<tr>
<th>Model</th>
<th>( k = \log n )</th>
<th>( k = \log^2 n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lower bound</td>
<td></td>
<td></td>
</tr>
<tr>
<td>asynchronous MP, SWCS, MWCS</td>
<td>( \frac{1}{n^{\Omega(1)}} )</td>
<td>( \frac{1}{n^{\Omega(1) \cdot \log n}} )</td>
</tr>
<tr>
<td>synchronous MP [20]</td>
<td>( \frac{1}{n^{\Omega(1) \cdot \log \log n}} )</td>
<td>( \frac{1}{n^{\Omega(1) \cdot \log n}} )</td>
</tr>
<tr>
<td>Upper bound</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWMR [8]</td>
<td>( \frac{1}{n^{\Omega(1)}} )</td>
<td>( \frac{1}{n^{\Omega(1) \cdot \log n}} )</td>
</tr>
<tr>
<td>MWMR [7]</td>
<td>( \frac{1}{n^{\Omega(1) \cdot \log \log n}} )</td>
<td>( \frac{1}{n^{\Omega(1) \cdot \log n}} )</td>
</tr>
</tbody>
</table>

The previous lower bound for the synchronous message-passing model [20] is \( q_k \geq \frac{1}{(ck)^k} \) for some constant \( c \). From the perspective of the expected total step complexity, given a nontermination probability \( \delta \), the lower bound of [20] implies \( \Omega((n - f) \log 1/\delta) \) steps, which is improved by our bound to \( \Omega((n - f) \log 1/\delta) \) steps.

In the shared-memory model with single-writer multireader registers, Aumann and Bender [8] show a consensus algorithm with probability \( 1 - \frac{1}{n^{\Omega(1)}} \) for terminating in \( O(n \log^2 n) \) steps. For multiwriter multireader registers, Aumann [7] presents an iterative consensus algorithm, with constant probability to terminate at each iter-
tion, independently of the previous iterations. This implies that the probability of
terminating after $k$ iterations is $1 - \frac{1}{c^k}$ for some constant $c$.

When agreement is required to hold only with high probability, Kapron et al. [27]
give an algorithm for the asynchronous message-passing model that always terminates
within $\text{polylog}(n)$ asynchronous rounds and has a probability $\frac{1}{\text{polylog}(n)}$ for disagreeing,
and an algorithm that always terminates within $2^{\Theta(\log^7 n)}$ asynchronous rounds and
has a probability $\frac{1}{\text{poly}(n)}$ for disagreeing.

An interesting open question is to close the gap between the values of the prob-
bability $\epsilon_k$ for disagreement achieved in the algorithms of [27] and the lower bounds
obtained in this paper on that probability. It is also interesting to tighten the bounds
in the synchronous model and for large values of $k$.

Our lower bounds can be used to estimate the error distribution and bound the
variance of the running time of randomized consensus algorithms. They do not yield
significant lower bounds for the expected step complexity—there is still a large gap
between the (trivial) lower bounds and upper bounds for the shared-memory model,
with a weak adversary.

Another broader research direction is to explore complexity bounds on random-
ized algorithms for other coordination problems, most notably, renaming [4] and set
consensus [16].

An alternative approach for deriving a lower bound on randomized consensus
under a weak adversary, suggested by Eli Gafni, is the following. First, derive a
lower bound for two processes in a message-passing system. Then, use the ABD
simulation [3] to simulate shared memory in a message-passing system, generalizing
the bound to hold for shared memory as well. Finally, use the BG simulation [12] to
extend the bound to $n$ processes. In this paper, we choose not to take this direction
for the following reasons. Proving the lower bound for two processes by showing an
indistinguishability chain with a certain size would be similar to section 3. Other
approaches, such as using topological arguments [26], require introducing a whole
different setting, which may not be simpler than the setting used in this paper. In
addition, the BG and ABD simulations incur a cost in the number of steps taken,
which may lead to a weaker bound than the one presented in this paper (recall from
section 4.2 the subtle considerations that allow using the reduction of single-writer
registers to multiwriter registers). However, we note that pursuing this approach is
appealing in the context of extending the lower bound to the set consensus problem
[16], as direct connectivity arguments [17] seem more involved than a topological
approach. We leave this for future research.

Acknowledgments. We would like to thank Valerie King for drawing our at-
tention to [20] and for interesting discussions, James Aspnes for coining the term
cheap-snapshot, and David Hay and Rotem Oshman for helpful comments. We also
thank the referees for helpful comments.

REFERENCES

pp. 165–176.
[3] H. Attiya, A. Bar-Noy, and D. Dolev, Sharing memory robustly in message-passing systems,
RANDOMIZED CONSENSUS UNDER A WEAK ADVERSARY


Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.


