Fast Randomized Consensus using Shared Memory

James Aspnes
School of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Maurice Herlihy
School of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213 *

September 17, 1996

Abstract

We give a new randomized algorithm for achieving consensus among asynchronous processes that communicate by reading and writing shared registers. The fastest previously known algorithm has exponential expected running time. Our algorithm is polynomial, requiring an expected $O(n^4)$ operations. Applications of this algorithm include the elimination of critical sections from concurrent data structures and the construction of asymptotically unbiased shared coins.

1 Introduction

A *consensus protocol* is a set of n asynchronous processors that communicate by applying operations to a shared object. The object may be a message channel, an array of read/write registers, or something more complex. Each process starts with an input value, either 0 or 1, and runs until it chooses a *decision value* and halts. A consensus protocol is correct if it is *consistent:* no two processes choose different decision values, *valid:* the decision value

*This research was sponsored by the Office of Naval Research (DOD), under contract N00014-88-K-0699. The work of the first author was supported by a National Science Foundation Graduate Fellowship. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

was some process's input value, and *wait-free:* each process decides after a finite number of steps.

Consensus protocols are interesting because they are fundamental to synchronization without mutual exclusion. The traditional approach to coordinating concurrent access to shared data objects is to rely on *critical sections*: only one process at a time is allowed to operate on the object. Nevertheless, critical sections are poorly suited for asynchronous, fault-tolerant systems: if a faulty process is halted or delayed in a critical section, non-faulty processes will also be halted or delayed. By contrast, an implementation of a concurrent data object is *wait-free* if it guarantees that any process will complete any operation in a finite number of steps, independent of other processes' halting failures or variations in speed. If there exists a consensus protocol for an object X, then one can use X to construct a wait-free implementation of any concurrent data object whose operations are total [21].

If the shared object X is an array of registers providing *read* and *write* operations, then consensus is known to be impossible [2, 12, 15, 21, 27]. If X is an array of registers providing *test-and-set* or *fetch-and-add* operations, then consensus is possible between two processes, but not among three [21, 27]. Nevertheless, in both cases, consensus among an arbitrary number of processes can still be achieved probabilistically. This paper presents two new randomized consensus protocols, one in which processes communicate by reading and writing shared registers, and one in which they communicate by applying *fetch-and-add* operations. The protocols are consistent, nontrivial, and they guarantee that each process decides after a finite *expected* number of steps. The only previously known read/write protocol, due to Abrahamson [1], requires an expected $2^{O(n^2)}$ operations. Ours is significantly faster, requiring an expected n^2 writes and n^4 reads. The *fetch-and-add* protocol requires an expected n^2 *fetch-and-add* operations.

The basic idea behind our protocols is quite simple. We first describe a simple protocol that has exponential expected running time if an adversary scheduler runs the processes in lockstep. Each process flips an unbiased coin at each round, and the protocol halts when all n processes simultaneously flip the "right" value. The probability of terminating at any particular round is $1/2^n$, so the expected number of rounds until termination is 2^n. A naive approach to speeding up the protocol is to replace the n independent coin flips with a single unbiased coin shared by the processes. Unfortunately, implementing an unbiased shared coin is provably impossible in an asynchronous system (see Section 8 below), so it would appear that no progress has been made. The key insight, however, is similar to one proposed by Chor, Merritt, and Shmoys [13]: it suffices to ensure that processes are sufficiently *likely* to flip the same value, and that an adversary scheduler has a sufficiently weak influence over which value is chosen. The heart of our consensus protocol is a *weak shared coin protocol* that guarantees: (1) processes are likely to observe the same outcome, (2) an adversary scheduler has only a weak influence over that outcome, and (3) the protocol has expected running time polynomial in the number of processes.

Consensus is often viewed as a game. One side, the processes, tries to achieve agreement against an adversary scheduler. The processes apply read and write operations to the shared registers, and the adversary chooses when the operations actually occur. Our adversary is extremely powerful: it has complete information about the processes' protocols, their internal states, and the state of the shared memory. It is not restricted to polynomial resources, and thus it cannot be outwitted by encryption schemes. The adversary cannot, however, predict future coin flips. Against such a powerful adversary, it may seem surprising that consensus can be achieved by a simple protocol in polynomial expected time.

2 Related Work

Fischer, Lynch, and Paterson [20] show that there is no consensus protocol for two processes that communicate by asynchronous messages. Dolev, Dwork, and Stockmeyer [15] and Dwork, Lynch, and Stockmeyer [16] give a comprehensive analysis of the circumstances under which consensus can be achieved by message-passing. Randomized protocols can achieve consensus when deterministic protocols cannot. Ben-Or [3] proposes a randomized consensus protocol with exponential expected running time that tolerates up to $n/5$ failures, where n is the number of processes. A consensus protocol due to Bracha and Toueg [7] relies on probabilistic properties of the message-passing system.

Loui and Abu-Amara [27] give several consensus protocols and impossibility results for processes that communicate through shared registers with various read-modify-write ("test-and-set") operations. Chor, Israeli and Li [12] give two randomized consensus protocols for shared read/write registers, one for two processes, and one for three processes. [1] Their protocols,

[1] The three-process protocol published in [12] has a bug: the termination condition must be strengthened to ensure consistency.

however, require a "strong" synchronization primitive: the ability to flip a coin and write the result in a single atomic step. By contrast, the protocols presented here, updating a register and changing process state are distinct transitions. Abrahamson [1] gives consensus protocols for both the "strong" model used by Chor, Israeli, and Li, and the more demanding "weak" model used here. As mentioned above, Abrahamson's consensus protocol for the weak model has exponential expected running time.

A number of protocols have been proposed for implementing shared coins in message-passing systems subject to byzantine or halting failures. (An excellent survey appears in [11].) Some constructions are direct [4, 8, 17], and others arise as parts of protocols for consensus [13], transaction commitment [14], or byzantine agreement [6, 13, 18, 33]. The models underlying these protocols differ from ours by assuming that private channels or encryption can be used to prevent the adversary from observing certain messages and processes' internal states. Chor and Coan [10] give a randomized byzantine agreement protocol that does not assume private communication, but restricts when the adversary may exploit knowledge of the processes' states.

3 Model

3.1 I/O Automata

Formally, we model processes and registers as I/O automata [28, 29]. An *I/O automaton* is a non-deterministic automaton A with the following components:

- *States*(A) is a finite or infinite set of states, including a distinguished set of starting states.
- *In*(A) is a set of *input events*,
- *Out*(A) is a set of *output events*,
- *Steps*(A) is a transition relation given by a set of triples (s', e, s), where s and s' are states and e is an event. Such a triple is called a *step*, and it means that an automaton in state s' can undergo a transition to state s, and that transition is associated with the event e.

If (s', e, s) is a step, we say that e is *enabled* in s'. I/O automata must satisfy the additional condition that inputs cannot be disabled: for each input event e and each state s', there exist a state s and a step (s', e, s).

An *execution* of an automaton A is a finite sequence $s_0, e_1, s_1, \ldots, e_n, s_n$ or infinite sequence $s_0, e_1, s_1, \ldots$ of alternating states and events such that s_0 is a starting state and each (s_i, e_{i+1}, s_{i+1}) is a step of A. A *history* of an automaton is the subsequence of events occurring in one of its executions. [2]

A new I/O automaton can be constructed by *composing* a set of I/O automata with disjoint output events. A state of the composed automaton S is a tuple of component states, and a starting state is a tuple of component starting states. The set of events of S, *Events*(S), is the union of the components' sets of events, and the set of output events of S, *Out*(S), is the union of the components' sets of output events. The sets of input events of S, *In*(S), is *Events*(S) − *Out*(S), all the events of S that are not output events for some component. A triple (s', e, s) is in *Steps*(S) if and only if, for all component automata A, one of the following holds: (1) e is an event of A, and the projection of the step onto A is a step of A, or (2) e is not an event of A, and A's state components are identical in s' and s. If H is a history of a composite automaton and A an automaton, $H|A$ denotes the subhistory of H consisting of events of A.

3.2 Processes, Registers, and Protocols

A *process* P is an I/O automaton with output events $\textsc{write}(P, v, R)$, $\textsc{read}(P, R)$, and $\textsc{decide}(P, v)$; input event $\textsc{return}(P, v, R)$; and internal event $\textsc{coin-flip}(P, x)$, where v is a value, R a register, and x (the *value* of the coin-flip) an element of the set {0,1}. The two COIN-FLIP events of a process represent possible results of a random decision made within the process; if either is enabled in a particular state, the other must also be enabled. A *register* R is an I/O automaton with input events $\textsc{write}(P, v, R)$ and $\textsc{read}(P, R)$, and the output event $\textsc{return}(P, v, R)$, where P is a process and v a value. A *protocol* $\{P_1, \ldots, P_n; R_1, \ldots, R_m\}$ is the I/O automaton composed by identifying in the obvious way the events for processes $P_1, \ldots, P_n$ and registers $R_1, \ldots, R_m$.

READ and WRITE events are called *invocations*, and RETURN events are called a *responses*. An invocation and response *match* if their process and register names agree. An invocation with no matching response is *pending*, and *complete*(H) is the maximal subsequence of H consisting only of invocations and matching responses. If H is a history, an *operation* of H is a pair

[2] To remain consistent with the terminology of [22], we use "event" where Lynch and Tuttle [29] use "operation" and "history" where they use "schedule."

consisting of an invocation and the next matching response. H is *sequential* if (1) the first event of H is an invocation, (2) each invocation, except possibly the last, is immediately followed by a matching response, and each response is immediately preceded by a matching invocation. A history that is not sequential is *concurrent.* A protocol history is *well-formed* if each $H|P_i$ is sequential, and a protocol is *well-formed* if each of its histories is well-formed. We assume all protocols are well-formed.

If we restrict our attention to sequential histories, then the behavior of a register can be specified in a particularly simple way: each value read was the last value written. We would like to ensure that a protocol's concurrent histories are "equivalent," in some sense, to its sequential histories. Formally, we capture this notion as follows. A protocol history H induces a partial "real-time" order $\prec_H$ on its operations: $op_0 \prec_H op_1$ if the response for op_0 precedes the invocation for op_1. Operations unrelated by $\prec_H$ are said to be *concurrent.* If H is sequential, $\prec_H$ is a total order. A protocol $\{P_1, \ldots, P_n; R_1, \ldots, R_m\}$ is *linearizable* [22] if, for each history H, there exists a sequential protocol history S such that:

- For all $P_i, H|P_i = S|P_i$.
- $\prec_H \subset \prec_S$

In other words, the history "appears" sequential to each individual process, and this apparent sequential interleaving respects the real-time precedence ordering of operations. A protocol is linearizable if all its histories are linearizable. Henceforth, we restrict our attention to linearizable registers, which are usually called *atomic registers* [26] in the literature. (A more general definition of linearizability, including comparisons with related correctness criteria, appears elsewhere [22].)

3.3 Randomization

The random non-determinism involved in the choice between COIN- FLIP$(P, 0)$ and COIN- FLIP$(P, 1)$ has a different nature from the "ordinary" non-determinism in the protocol. It is possible to make the distinction formally by placing the other non-deterministic choices under the control of an *adversary*, a function A which maps each of the protocol's finite executions $s_0, e_1, s_1, \ldots, e_n, s_n$ to a set of events enabled in s_n, such that for all finite executions ξ, $A(\xi)$ consists either of a single non-COIN-FLIP event or a pair of COIN-FLIP events representing the two possible outcomes of a COIN-FLIP at some process.

The intent is that the adversary controls which executions are possible. More formally, we say that an adversary A *permits* a (possibly infinite) execution $\xi = s_0, e_1, s_1, \ldots$, if, for every event e_i in ξ, e_i is an element of $A(s_0, e_1, s_1, \ldots, s_{i-1})$. We write Ξ_A for the set of executions that A permits, and $\Xi_{A,s}$ for the subset of Ξ_A consisting of executions which have s as their initial state.

Let $c(\xi)$ be the sequence of coin-flip values in ξ. It is a straightforward consequence of the constraints on the domain of an adversary function A that, for each countable sequence C of 0's and 1's, there exists exactly one $\xi_{A,s}(C)$ in $\Xi_{A,s}$ such that $c(\xi_{A,s}(C))$ is a prefix of C. We can think of $\xi_{A,s}$ as a measurable function from the sequence space Ω on the set $\{0,1\}$ to the sequence space Ξ of protocol executions [23]. We can thus use $\xi_{A,s}$ to define a probability measure on Ξ by transforming the probability measure on Ω as follows:

$$P_{A,s}(X) = P(\{C|\xi_{A,s}(C) \in X\})$$

(where the probability on the left is defined only when $X \subset \Xi$ is measurable Ξ). An immediate consequence of the definition is that $P_{A,s}(X) = 0$ for any X which is disjoint from $\Xi_{A,s}$.

For now, we leave the probabilities associated with coin flips as an unspecified parameter of the model.

3.4 Consensus Protocols

A *consensus protocol* is a protocol whose processes each have two initial states, corresponding to input values of 0 or 1, respectively, and whose histories all satisfy the following conditions:

1. *Consistency.* Every DECIDE event in the history has the same value, which must be an input value for at least one of the processes.

2. *Termination* A DECIDE event for P must be the last output event of P.

3. *Validity.* If s is an input state in which some processes start with different values, there exist adversaries A, B such that $P_{A,s}(\{\xi|\xi$ contains DECIDE$(P, 0)$ for some $P\})$ and $P_{B,s}(\{\xi|\xi$ contains DECIDE$(P, 1)$ for some $P\})$ are both non-zero.

The first condition guarantees that the protocol actually achieves consensus. The second condition is not critical to describing a consensus protocol, but is necessary for identifying when a protocol is finished. The third condition excludes protocols which achieve consensus trivially by fixing the outcome in advance.

The *running time* $r(H)$ of a protocol history H is the length of the shortest prefix of H which contains a DECIDE event for every process in the protocol. The *worst-case expected running time* of a protocol is given by:

$$\max_{A,s} \sum_{i=0}^{\infty} i P_{A,s}(\{\xi | r(\xi) = i\})$$

which is simply the expected running time of the protocol against the worst possible adversary. If, for some adversary A and initial state s, the sum in the above expression does not converge, we say that the protocol has an infinite worst-case expected running time.

4 An Exponential Consensus Protocol

Each process P has a register with two fields:

- `prefer`, if distinct from $\perp$, is the value P would choose if it were to complete the protocol executing in isolation,
- `round` is a counter that keeps track of the number of rounds P has executed so far.

A process Q *agrees* with P if (1) both `prefer` fields are equal, and (2) neither is $\perp$. A process is a *leader* if its `rounds` field is greater than or equal to any other process's `rounds` field.

The protocol, shown in Figure 1, works as follows. Initially, P's register is initialized so that `round` is 0 and `prefer` is $\perp$. Process P starts by setting `round` to 1, and `prefer` to its input value. P then enters the main loop of the protocol. It reads all processes' registers. The protocol terminates if P is a leader, and if all processes whose `round` fields trail P's by less than two agree with P. Otherwise, if the leaders agree, P updates its register to agree with the leaders, increments its `round` counter, and resumes the loop. Otherwise, if its `prefer` field is not $\perp$, P "warns" the other processes that it may change its preference by setting `prefer` to $\perp$ before resuming the loop. If `prefer` is already $\perp$, then P chooses a new preference by an unbiased coin flip, increments `round`, and resumes the loop.

Although the `rounds` field is potentially unbounded, larger values are reached with lower probabilities, thus the likelihood of overflow can be made arbitrarily small.

```
% Initially:
  r.prefer := BOTTOM    % preferred value
  r.round := 0 % racing counter

% The algorithm
r := [prefer: input, round: 1]
while true do
  read registers
  if all who disagree trail by 2 AND I'm a leader
    then decide(r.prefer)
  % Agree with unanimous leaders ...
  elseif leaders agree then
    r := [prefer: leader.prefer, round: r.round + 1]
  % Warn of impending change
  elseif r.prefer ~= BOTTOM then
    r := [prefer: BOTTOM, round: r.round]
  % Guess a new value.
  else r := [prefer: flip(), round: r.round + 1]
  end % if
  end %for
```

Figure 1: An Exponential Consensus Protocol

4.1 Consistency

Let H be a history (sequence of reads, writes, and flips) permitted by a particular adversary. For brevity, we say that process P *prefers* $v \neq \perp$ at round r if P writes `[prefer: v, round: r]` at some step in H, and that P is *busy* at round r if it writes `[prefer: ⊥, round: r]`. The *first* process to prefer v at round r is the one whose write occurred earliest in H. We use v and v' to stand for the two distinct decision values.

Lemma 1 *If P prefers v at round r and v' at round $r+1$, then some $Q \neq P$ prefers v' at round $r' \geq r$, and Q's write of v' precedes P's write of v'.*

Proof: P can change preference from v to v' in one of two ways: if it observes that all leaders agree on v', or if it observes that the leaders disagree. In either case, some other Q prefers v' at round $r' \geq r$, and since P read that value, Q's write of v' must precede P's. ■

Lemma 2 *If every process that completes round r in H prefers v at that round, then no process prefers a distinct value at any higher round.*

Proof: Suppose not. Let P be the first process in H to prefer v' at round $r' > r$. Lemma 1 implies that some Q prefers v' at round $r'' \geq r$. Since it is given that all processes that completed round r prefer v, it follows that $r'' > r$, contradicting the hypothesis that P is the first process to switch its preference after round r. ■

By similar reasoning:

Lemma 3 *If every process that completes round r in H prefers v at that round, then no process is busy at any higher round.*

Lemma 4 *If every process that completes round r in H prefers v at that round, then no process completes round $r+2$ without deciding v.*

Proof: By contradiction. Any process that decides after round r must decide v, since Lemma 2 implies it must prefer that value. Let Q be the first process to fail to decide at round $r+2$. Since all earlier processes to start that round have decided, Q is a leader. If Q fails to decide, then it must disagree with another process P at rounds r or $r+1$. Either P prefers $v' \neq v$ at that round, contradicting Lemma 1, or P is busy at that round, contradicting Lemma 3. ■

Lemma 5 *If P decides v after writing round r in H, then no other process prefers v' at round r.*

Proof: Suppose not. Let Q be the first process to prefer v' at round r in H. There are two cases to consider, depending on Q's preference at round $r-1$.

Case 1: Q prefers v at round $r-1$. We claim that Q can change its preference only as a result of a coin flip. If Q switched preference between rounds $r-1$ and r without flipping, then it must have observed that the leaders prefer v'. Let H' be the prefix of H strictly preceding Q's write. Because all processes that completed round r in H' prefer v, these leaders cannot be at a round greater than r (Lemma 2). Because Q is the first process to prefer v' at round r (hypothesis), these leaders cannot be at round r. But Q itself prefers v at round $r-1$, therefore the leaders cannot agree.

Before Q can flip a coin, however, it must set its `prefer` field to $\perp$ and reread the registers. Q now observes that P prefers v at round r. No other process prefers v' in H' at round r (hypothesis) or higher (Lemma 2). No process is busy at round r or higher, since the first such process must have observed another process that prefers v' at round r or higher. Therefore, Q observes that the leaders agree on v, and it resets its register to agree with the leaders, contradicting the hypothesis.

Case 2: Q prefers v' at round $r-1$. Since P decided v, it must have read Q's register at round $r' < r-1$. Before Q can advance to round r, it must reread P's register, observing that P prefers v at round r. By an argument essentially identical to the one given above, no process disagrees with P at levels r or above, a contradiction. ■

Theorem 6 *This consensus protocol is consistent.*

Proof: If any process decides on v after writing round r, then all processes will prefer v at round r (Lemma 5), and hence all processes will eventually decide v (Lemma 4). ■

This protocol can be extended to allow decision values from an arbitrary domain, not just {0,1}. Before joining the protocol, each process writes its initial value to a public register. Instead of flipping a coin to change preference, a process randomly adopts a leader's preference.

4.2 Running Time

A process is *deterministic* at round r if it does not flip a coin at that round, and *non-deterministic* otherwise. Let V and V' be the sets of processes that respectively prefer v and v' at round r.

Lemma 7 *The set of non-deterministic processes at round r encompasses at least one of V and V'.*

Proof: We show that if P and Q belong to V and V', then at least one of the two must be non-deterministic.

Let P be the first process to write a preference, say v, at round r. If P is non-deterministic, then it must have observed that the leaders prefer v. Since P is the first process to enter round r, it must have observed that all processes at round $r-1$ prefer v. Let Q be the first process to prefer v' at round r. If Q is deterministic, then it must have observed that the leaders prefer v'. Since all processes at rounds r and higher prefer v, Q must have observed that all processes at round $r-1$ prefer v'. Each process, however, writes out its preference for round $r-1$ before reading the other's register, thus at least one of the two must have observed a disagreement before entering round r, and that process must be non-deterministic. ■

Lemma 7 implies that the deterministic processes at round r have the same preference. We now show that if the non-deterministic processes choose the same preference, then the adversary cannot force the determinstic processes to disagree.

Lemma 8 *Let v be the first value written at round $r-1$. If the non-deterministic processes at round r all choose v, then all processes have the same preference at round r.*

Proof: The result is immediate if there are no deterministic processes. Suppose P is the first deterministic process to prefer v' at round r. Since P is deterministic, it must have observed that the leaders prefer v'. The leaders could not have been at round r or higher, since all such processes are non-deterministic, and they prefer v both at round r (by hypothesis) and at higher rounds (Lemma 2). P observed at least one process at round $r-1$, namely itself. Since P's write at round $r-1$ precedes any of its reads, P must also have read any values whose writes at that round precede its own, including the earliest. ■

Notice that the first value written at round $r-1$ is fixed before the first process begins round r, and therefore the adversary cannot force disagreement by allowing the non-deterministic processes to choose their preference, and then somehow forcing the deterministic processes to disagree.

The running time of the consensus protocol thus depends primarily on the degree of control the adversary can exercise over the outcomes of coin flips.

Definition 9 *Let $H_{A,r}$ be strict lower bound on the probability that no process flips* tails *at round r when running against adversary A, and let $T_{A,r}$ be defined symmetrically. The* defiance probability *δ is:*

$$\delta = \min_{A,r}(H_{A,r}, T_{A,r}).$$

Informally, δ is the probability that all non-deterministic processes will flip a given value given that the adversary "wants" at least one process to flip the other value.

Theorem 10 *The consensus protocol has worst-case expected running time $O(1/\delta)$ rounds.*

Proof: Consider the set of coin flips associated with each round after the first. If all processes are deterministic, then they have identical preferences (Lemma 7), and the protocol is about to terminate (Lemma 4). If some processes are non-deterministic, then the protocol terminates if they all choose the first value written at the previous round (Lemma 8). Since this value is fixed before any process performs a coin flip at that round, the protocol terminates at that round with probability greater than or equal to δ. The protocol thus acts like a Bernoulli process, where the probability of terminating at each round is at least δ, and the expected running time is at most $1/\delta$. ■

Corollary 11 *The consensus protocol has a worst-case expected running time of $O(n^2/\delta)$ steps.*

Proof: In each round, each process performs at most $2n$ READ operations, one COIN-FLIP, and two WRITE operations, for a total of $2n+3$ steps. Thus the total number of steps taken per round by all processors is $O(n^2)$, giving a maximum total running time of $O(n^2/\delta)$ steps. ■

Corollary 12 *If processes flip independent unbiased coins, then δ is $1/2^n$, and the protocol has a worst-case expected running time of $O(2^n)$ steps.*

This bound is easily seen to be tight. The adversary can run the processes in lockstep, so that all n processes observe disagreement at each round, and all flip to choose a preference for the next round.

5 The Weak Shared Coin Protocol

As noted above, the protocol runs for exponential expected time if processes flip unbiased, independent coins. In this section we transform the exponential protocol into a protocol in which processes reach agreement after an expected $O(n^2)$ writes and $O(n^4)$ reads. The basic idea is to have the processes undertake a *weak shared coin* protocol that, in essence, simulates a coin shared by all processes. The weak shared coin protocol is parameterized by a value $K > 1$. It has the key property that any adversary has only a weak influence over the protocol's outcome:

Definition 13 *A weak shared coin protocol has defiance probability $(K-1)/2K$.*

The influence that can be exercised by any adversary is thus independent of n, and asymptotically approaches zero as K increases. We emphasize that the protocol does not guarantee that all processes observe the same outcome, only that they do so with probability at least $(K-1)/2K$.

Corollary 14 *If processes flip a weak shared coin at each round, then the consensus protocol terminates in an expected $O(1)$ rounds.*

The complexity of achieving consensus in terms of primitive reads and writes is thus the complexity of implementing the weak shared coin.

5.1 Implementing a Weak Shared Coin

The weak shared coin protocol is implemented using a *shared counter* abstraction, whose implementation in terms of *reads* and *writes* is given in Section 6. The counter is linearizable with the following sequential specification:

```
    inc = proc(c: counter)
```

increments the counter,

```
    dec = proc(c: counter)
```

decrements the counter, and

```
    readCounter = proc(c: counter) returns (int)
```

returns the counter's current value.

The weak shared coin protocol is shown in Figure 2. The processes collectively undertake a random walk: each process flips an unbiased coin, and depending on the outcome, increments or decrements the shared counter. It then reads the counter. If the observed value is greater than or equal to Kn, the process decides *heads*, and if the observed value is less than or equal to $-Kn$, it decides *tails*. Informally, the only way the adversary can influence the outcome of the protocol is to suspend processes that are about to move the counter in the undesired direction. After suspending $n-1$ such processes, however, the adversary has "used up" its influence, and the remaining process is free to wander at random. As K increases, the importance of this bias decreases.

Let H and T be the respective number of *heads* and *tails* generated so far.

Lemma 15 *If $H - T < -(K+1)n$ then all undecided processes will eventually decide* tails*.*

Proof: Since the adversary can suspend at most one write per process, the counter value read by any process can differ from $H-T$ by at most $n-1$. Once $H-T$ falls below $-(K+1)n$, every process that samples the counter will observe a value less than or equal to $-Kn$. ■

By similar reasoning:

Lemma 16 *If some process decides* heads*, then at the time of its last read, $H - T > (K-1)n$.*

We can combine these two observations to derive a bound on the likelihood the adversary can force disagreement, or a desired outcome.

Theorem 17 *The adversary can force some process to flip* heads *with probability at most $(K+1)/2K$.*

Proof: Lemma 16 implies that no process can decide *heads* before $H-T$ reaches $(K-1)n$ for the first time. If, however, $H-T$ falls below $-(K+1)n$ before reaching $(K-1)n$, then Lemma 15 implies that no process can decide *heads*. If we make the conservative assumption that the adversary can force some undecided process to choose *heads* if $H-T \geq (K-1)n$, then the value of $H-T$ can be viewed as a random walk starting at the origin with absorbing barriers at $-(K+1)n$ (all decide *tails*) and $(K-1)n$ (some may decide *heads*). It is a standard result of random walk theory [19, Ch. XIV] that the probability of reaching $(K-1)n$ before $-(K+1)n$ is $(K+1)/2K$. ■

One way to make certain that some process flips *heads* is to force two processors to disagree.

Corollary 18 *The adversary can force processes to disagree with probability less than or equal to* $(K-1)/2K$.

Theorem 19 *The worst-case expected running time of the weak shared coin protocol is* $O(n^2)$ *rounds.*

Proof: Instead of promoting a particular outcome, suppose the adversary adopts a dilatory strategy, seeking to prolong the protocol for as long as possible. As noted above, the protocol will terminate whenever the absolute value of the counter exceeds $(K+1)n$, thus the protocol behaves like a random walk starting at the origin with absorbing barriers at $(K+1)n$ and $-(K+1)n$. It is a standard result of random walk theory [19] that the expected running time of such a walk is $(K+1)^2n^2$, i.e. $O(n^2)$. ■

6 The Counter Abstraction

The counter implementation is a straightforward adaptation of an algorithm proposed by Lamport [25] for read/write registers. The counter is represented by an n-element array of registers, one for each process. Each register has two fields: a `count` field incremented whenever that process alters the register's value, and a `val` field representing that process's contribution to the current counter value. To increment or decrement the counter, P overwrites its register with a new value whose `count` field is incremented, and whose `val` field is incremented or decremented. To read the counter it scans the array twice: if both scans yield identical values, the read returns the sum of the `val` fields, otherwise the read is restarted.

```
flip = proc(coin: counter) returns (bool)
  while true do
    if flip() then inc(coin) else dec(coin) end
    state := readCounter(coin)
    if state >= K*N then return (heads)
      elseif state <= -K*N then return (tails)
      end
    end
  end flip
```

Figure 2: The Weak Shared Coin Protocol

Theorem 20 *The counter implementation is linearizable.*

Proof: The update operations, i.e., increments and decrements, are totally ordered in a natural way: $I_0 < I_1$ if I_0 writes its array element first. A *readCounter* operation *observes* an update if the former reads an array element after the latter writes it. A *readCounter* returns the sum of the updates it observes. It suffices to show that *readCounter* implements a "snapshot" — if I_0 ¡ I_1, and a *readCounter* operation R observes I_1, then it also observes I_0. Define R's *first (second) read* of I_0 be its penultimate (last) read of the array element written by I_0, and similarly for the other operations. Because $I_0 < I_1$, I_0's write precedes I_1's write. Because R observes I_1, and the array value doesn't change between R's first and second reads of I_1, I_1's write precedes R's first read of I_1. Because R starts the second scan only after finishing the first, R's first read of I_1 precedes its second read of I_0, and therefore R observes I_0. ■

Note that the *inc* and *dec* operations are wait-free, but *readCounter* can be starved if it is interrupted by an infinite sequence of writers. The adversary canot exploit this property to force the protocol to run forever: after enough writes, the next reader will drop out of the protocol, and because there are only finitely many processes, it will eventually be possible for some process to complete a read.

Lemma 21 *While the weak shared coin protocol is running, the n processes together cannot execute more than $2n^2$ primitive reads or writes without* *incrementing or decrementing the shared counter.*

Proof: An increment or decrement completes after a single primitive write. Any *readCounter* that does not overlap an increment or decrement completes after $2n$ primitive reads. Thus, the n processes together can execute $2n^2$ primitive reads before being forced to increment or decrement the counter. ■

Theorem 22 *If processes flip a weak shared coin at each round, then the consensus protocol requires an expected $O(n^2)$ primitive writes and $O(n^4)$ primitive reads.*

Proof: Each *inc* or *dec* translates into to a single write, so the $O(n^2)$ expected steps needed to exhaust the random walk translate into an expected $O(n^2)$ primitive write operations. Lemma 21 implies that each increment or decrement requires $O(n^2)$ primitive reads, resulting in an expected $O(n^4)$ primitive reads. ■

7 Consensus Using Fetch-And-Add

The *fetch-and-add* operation [24] atomically adds a quantity to a register and returns the register's old value. *Fetch-and-add* solves consensus deterministically for two processes, but not for three or more [21]. Figure 4 shows a weak shared coin implementation using *fetch-and-add* operations. Not surprisingly, *fetch-and-add* is more efficient than *read* and *write*; it is straightforward to show that this protocol completes in an expected $O(n^2)$ total operations.

8 Strong Shared Coin Protocols

A *strong shared coin protocol* is a consistent wait-free algorithm by which n processes agree on a value in $\{$*heads*, *tails*$\}$ by applying operations to a shared object. A shared coin protocol is *unbiased* if both choices are equally likely; i.e., the adversary has no control over the outcome. A naive solution might have each process flip an unbiased local coin to choose its input value, and then achieve consensus with the others. Such a solution is heavily biased, however, since an adversary that "wants" an outcome of *heads* can run only the processes that prefer *heads*. Against such an adversary, this naive

```
reg = record[count: int, val: int] % Initially [0,0]

Inc = proc(counter: array[reg])
  r: reg := counter[self]
  counter[self] := [r.count + 1, r.val + 1]
  end Inc

Dec = proc(counter: array[reg])
  r: reg := counter[self]
  counter[self] := [r.count + 1, r.val - 1]
  end Dec

ReadCounter = proc(counter: array[reg]) returns (int)
  scan1, scan2: array[int]
  while true do
    for i: in 1..n do
      scan1[i] := counter[i]
      end
    for i: in 1..n do
      scan2[i] := counter[i]
      end
    if scan1 = scan2 then return (sum(scan1)) end
    end
  end ReadCounter
```

Figure 3: Implementation of Shared Counter

```
flip = proc(coin: register) returns (bool)
  while true do
    if flip()
      then state := fetch-and-add(coin, 1)
      else state := fetch-and-add(coin, -1)
      end
    if state >= K*N then return (heads)
      elseif state <= -K*N then return (tails)
      end
    end
  end flip
```

Figure 4: Weak Shared Coin Protocol using Fetch-And-Add

protocol will decide *tails* only if all processes initially flip *tails*, a probability of $1/2^n$.

Theorem 23 *An unbiased strong shared coin protocol is impossible.*

Proof: By contradiction. For any protocol, we construct an adversary that produce *heads* with probability greater than 1/2. Assume we have an unbiased protocol, and let P and Q be any two processes. Define a configuration's *range* to be the set of probabilities of eventually deciding heads for all possible adversaries. Define a process's current preference as the probability that it will eventually decide *heads* if it is run uninterrupted until deciding. Note that each process's preference must appear in the range, as running that process without interruption is a possible behavior of an adversary. For an unbiased protocol, the initial configuration's range is $\{1/2\}$, and thus each process's preference is 1/2.

Consider the following adversary. Run P until it is about to take a step that changes the current range. Such a step must eventually occur, because the protocol cannot run forever. Moreover, the step must be a coin flip internal to P, since all other steps are deterministic and under the adversary's control. Before the coin flip, the configuration's range is $\{1/2\}$, and the preference of both P and Q is 1/2. Suppose P's local flip yields *heads* with probability h. Let r_h (r_t) be the range resulting if P flips *heads* (*tails*). Let A be an adversary corresponding to some element a_h of r_h not equal to 1/2 and let a_t be an element of r_t yielded by A. Then since the protocol is unbiased, we have

$$1/2 = h \cdot a_h + (1-h) \cdot a_t$$

implying that one of a_h, a_t is greater than 1/2 and the other less. Assume $a_h > 1/2 > a_t$; the other case is symmetric. Since Q cannot directly observe P's coin flip, its preference continues to be 1/2.

If the outcome of P's flip is *heads*, then the adversary can ensure an outcome of *heads* with probability a_h by emulating A. If the outcome of P's flip is *tails*, the adversary can run Q uninterruptedly until it decides, ensuring an outcome of *heads* with probability 1/2. Taken together, the adversary can ensure *heads* with probability:

$$h \cdot a_h + (1-h)/2$$

Since $a_h > 1/2$, however, this quantity exceeds 1/2, contradicting the hypothesis that the protocol is unbiased. ■

Note that this proof makes no assumptions about *how* processes communicate; they could use read/write registers, *fetch-and-add* registers, messages, or other objects.

Although we have shown that the adversary can always introduce some bias, we have given no indication of how large that bias may be. A shared coin protocol is *asymptotically unbiased* if the bias introduced by the adversary can be made arbitrarily small.

Theorem 24 *An asymptotically unbiased strong shared coin protocol with expected running time polynomial in the number of processes is possible using shared read/write registers.*

Proof: Have each process choose *heads* or *tails* using a weak shared coin, and then run the polynomial consensus protocol given above. The adversary can influence the outcome by biasing the initial preferences. If any process prefers *heads*, the adversary can suspend the others, while if all processes prefer *tails*, the adversary has no more control. The likelihood the adversary can force an outcome of *heads* in the initial round is thus $(K+1)/2K$, which approaches 1/2 as K increases. ■

9 Discussion

Most recent work on wait-free synchronization has focused on the construction of *atomic read/write registers* [5, 9, 25, 26, 30, 31, 32, 34]. Starting with "safe" bits for which overlapping read and write operations have unpredictable effects, these papers describe a sequence of algorithms for constructing wait-free implementations of read/write registers providing successively stronger guarantees, culminating in algorithms that permit multiple concurrent readers and writers, an impressive achievement.

Nevertheless, reading and writing to individual registers is not the level of abstraction at which most programs are written. Wait-free synchronization will be useful in practice only if it is possible to construct wait-free implementations of objects with richer semantics than registers, objects such as *test-and-set* registers, stacks, queues, file system directories, databases, etc. It is known, however, that atomic read/write registers have few, if any, interesting applications in this area [21]. Using atomic read/write registers, it is impossible to construct a wait-free implementation of: (1) common data types such as sets, queues, stacks, priority queues, or lists, (2) most if not all the classical synchronization primitives such as *test-and-set*, *compare-and-swap*, and *fetch-and-add*, and (3) such simple memory-to-memory operations as *move* or memory-to-memory *swap*.

One way to interpret these impossibility results is that atomic read/write registers are a computational dead-end, and that wait-free synchronization is unrealizable by machine architectures in which processes communicate by reading and writing shared memory locations. The results in this paper suggest an alternative position. If one can achieve consensus, one can transform a sequential implementation of any object whose operations are total (i.e., defined in every state) to a wait-free linearizable implementation [21], where each operation requires at most n rounds of consensus. In the same way, the randomized consensus protocol presented here can be used to transform any sequential object implementation into a randomized wait-free implementation, where each operation has expected running time polynomial in the number of processes. In short, wait-free synchronization is indeed realizable under conventional architectures, provided the wait-free guarantee is probabilistic in nature.

Acknowledgments

Hagit Attiya, Nir Shavit, and the anonymous referees provided many useful comments.

References

[1] K. Abrahamson. On achieving consensus using a shared memory. In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1988.

[2] J.H. Anderson and M.G. Gouda. The virtue of patience: Concurrent programming with and without waiting. Private Communication.

[3] M. Ben-Or. Another advantage of free choice: completely asynchronous agreement protocols. In *Second ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 27–30, August 1983.

[4] M. Ben-Or and N. Linial. Collective coin flipping, robust voting schemes, and minima of banzhaf values. In *Twenty-sixth Annual Symposium on Foundations of Computer Science*, pages 408–416, October 1985.

[5] B. Bloom. Constructing two-writer atomic registers. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 249–259, 1987.

[6] G. Bracha. An o(log n) expected rounds randomized byzantine generals algorithm. In *Seventeenth Annual Symposium on Theory of Computation*, 1985.

[7] G. Bracha and S. Toueg. Resilient consensus protocols. In *Second ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 12–26, August 1983.

[8] A. Broder and D. Dolev. Flipping coins in many pockets (byzantine agreement on uniformly random values. In *Twenty-Fifth Annual Symposium on Foundations of Computer Science*, pages 157–170, October 1984.

[9] J.E. Burns and G.L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 222–231, 1987.

[10] B. Chor and B. Coan. A simple and efficient randomized byzantine agreement algorithm. *IEEE Transactions on Software Engineering*, SE-11(6):531–539, June 1985.

[11] B. Chor and C. Dwork. *Randomization in Byzantine Agreement*, volume 4. JAI Press, 1987.

[12] B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 86–97, 1987.

[13] B. Chor, M. Merritt, and D.B. Shmoys. Simple constant-time consensus protocols in realistic failure models. In *Proceedings of the Fourth ACM Symposium on Principles of Distributed Computing*, pages 152–160, 1985.

[14] B. Coan and J. Lundelius. Transaction commit in a realistic fault model. In *Fifth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 40–52, August 1986.

[15] D. Dolev, C. Dwork, and L Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[16] C. Dwork, N. Lynch, and L Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):228–323, April 1988.

[17] C. Dwork, D. Shmoys, and L. Stockmeyer. Flipping persuasively in constant expected time. In *Twenty-Seventh Annual Symposium on Foundations of Computer Science*, pages 222–232, October 1986.

[18] P. Feldman and S. Micali. Optimal algorithms for byzantine agreement. In *Twentieth Annual ACM Symposium on Theory of Computing*, pages 148–161, May 1988.

[19] W. Feller. *An Introduction to Probability Theory and its Applications*, volume 1. John Wiley and Sons, 1957.

[20] M. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed commit with one faulty process. *Journal of the ACM*, 32(2), April 1985.

[21] M.P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1988.

[22] M.P. Herlihy and J.M. Wing. Axioms for concurrent objects. In *14th ACM Symposium on Principles of Programming Languages*, pages 13–26, January 1987.

[23] J.G. Kemeny, J.L. Snell, and A.W. Kapp. *Denumerable Markov Chains.* D. Van Nostrand, 1966.

[24] C.P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization on multiprocessors with shared memory. In *Fifth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1986.

[25] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[26] L. Lamport. On interprocess communication, parts i and ii. *Distributed Computing*, 1:77–101, 1986.

[27] M.C. Loui and H.H. Abu-Amara. *Memory Requirements for Agreement Among Unreliable Asynchronous Processes*, volume 4, pages 163–183. JAI Press, 1987.

[28] N.A. Lynch and M. Merritt. Introduction to the theory of nested transactions. Technical Report MIT/LCS/TR-387, M.I.T. Laboratory for Computer Science, April 1986.

[29] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR-387, M.I.T. Laboratory for Computer Science, April 1987.

[30] R. Newman-Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 232–249, 1987.

[31] G.L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

[32] G.L. Peterson and J.E. Burns. Concurrent reading while writing ii: the multi-writer case. Technical Report GIT-ICS-86/26, Georgia Institute of Technology, December 1986.

[33] M. Rabin. Randomized byzantine generals. In *Twenty-fourth Annual Symposium on Foundations of Computer Science*, pages 403–409, October 1983.

[34] A.K. Singh, J.H. Anderson, and M.G. Gouda. The elusive atomic register revisited. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 206–221, August 1987.