Wait-Free Consensus with Infinite Arrivals

James Aspnes*
aspnes@cs.yale.edu

Gauri Shah*
gauri.shah@yale.edu

Jatin Shah
jatin.shah@yale.edu

Department of Computer Science
Yale University
New Haven, CT 06520-8285, USA

ABSTRACT

A randomized algorithm is given that solves the wait-free consensus problem for a shared-memory model with *infinitely many* processes. The algorithm is based on a weak shared coin algorithm that uses weighted voting to achieve a majority outcome with at least constant probability that cannot be disguised even if a strong adversary is allowed to destroy infinitely many votes. The number of operations performed by process i is a polynomial function of i. Additional algorithms are given for solving consensus more efficiently in models with an unknown upper bound b on concurrency or an unknown upper bound n on the number of active processes; under either of these restrictions, it is also shown that the problem can be solved even with infinitely many *anonymous* processes by prefixing each instance of the shared coin with a naming algorithm that breaks symmetry with high probability. For many of these algorithms, matching lower bounds are proved that show that their per-process work is nearly optimal as a function of i, b, or n. The case of n active processes gives an algorithm for anonymous, *adaptive* consensus that requires only $O(n \log^2 n)$ per-process work, which is within a constant factor of the best previously known *non-adaptive* algorithm for a strong adversary. Finally, it is shown that standard universal constructions based on consensus continue to work with infinitely many processes with only slight modifications. This shows that in infinite distributed systems, as in finite ones, with randomness all things are possible.

1. INTRODUCTION

There has been much recent interest in the distributed computing community in algorithms that support arbitrarily many participants. A natural way to study such algorithms is to define a system in which infinitely many processes may join the protocol over time, and see what problems remain solvable in such a system. Recent work by Gafni *et al.* [20] has shown that wait-free renaming, collect, and snapshot algorithms exist for infinitely many processes, showing that many of the fundamental building blocks of shared-memory distributed algorithms do not require a bound on the number of processes.

In this paper, we give randomized wait-free algorithms that allow infinitely many processes to solve the much more difficult *consensus problem* [24] under various conditions. Since the consensus problem is known to be universal for constructing wait-free data structures [22], these algorithms show that infinite arrivals are not in themselves an obstacle to building wait-free algorithms.

1.1 Model

Our model follows that used in previous work on systems that support infinite arrivals [20], with some additions to incorporate randomization.

The basic model assumes a countably infinite collection of processes that communicate by means of a countably infinite collection of atomic read-write registers, each of which can store arbitrarily large finite values.[1] The reason for assuming infinitely many processes is to model *infinite arrivals*, where the number of processes present in the system at any time is finite, but where this quantity can grow arbitrarily large over time. Having accepted unboundedly many processes, we must also provide unboundedly many registers, due to the $\Omega(\sqrt{n})$ lower bound on the space needed for n-process consensus proved by Fich *et al.* [18].

We model concurrency by interleaving. A *schedule* consists of a sequence of events, each of which is either an atomic read operation, an atomic write operation, or an internal coin-flip of some process. Read and write operations have the usual effects on the shared memory. Coin-flips are random transitions internal to a process whose outcome cannot be predicted by the adversary. All events are treated as occurring sequentially, with each event transforming a pre-

*Supported by NSF grants CCR-9820888 and CCR-0098078.

STOC'02, May 19-21, 2002, Montreal, Quebec, Canada.
Copyright 2002 ACM 1-58113-495-9/02/0005 ...$5.00.

[1] Most of our algorithms do not extend to uncountable collections of processes; however, since we assume only finitely many processes arrive at each finite time, having more than countably many processes participate in a single execution would require an execution involving uncountably many steps. Although the study of executions taking uncountable time might be of some mathematical interest, the problem of defining the state of the system at limit points fills us with dread, and there are likely to be significant complications to probabilistic analysis with uncountable sets of events. Hence our assumption of countability throughout.

vious global state into a new global state. The occurrence of an event is called a *step*. In any given state there may be many events that may occur; such events are called *pending*.

An *adversary* chooses which pending event occurs at each step; this adversary may be *strong*, in which case it can base this decision on the entire state of the system, including the contents of registers and internal states of the processes; or *weak*, in which case it must choose equivalent pending events in states that satisfy some given equivalence relation. Our weak adversary will be the *content-oblivious* adversary of [15], which cannot observe contents of registers, values of pending write operations, or internal states of the processes. Both strong and weak adversaries are defined by their *strategies*, functions from partial executions to pending events.

Every process has two special events, which are used to keep track of the number of processes that are running concurrently. The first event of any process is a *birth event*; such an event corresponds to the process waking up and beginning its computation. A process may also experience a *death event*, corresponding to either termination or a crash failure. A process that has performed a birth but not a death is *live*. For some of our results we assume a bound on the number of live processes in any state.

Birth events are enabled only in the initial state of a process; once born, a process may die, but cannot be born again. Death events are always enabled for any live process. However, we require that an adversary strategy yields with probability 1 at least one process that, without dying, either terminates by reaching a state in which it has no pending operations, or executes infinitely many operations.

Because we have infinitely many processes, we cannot expect to bound the total work performed by any algorithm. So we will concentrate instead on the per-process work, the total number of read and write operations performed by each process. A protocol is *randomized wait-free* if the per-process work is finite with probability 1 for all processes that execute the protocol. We will only be interested in randomized wait-free protocols.

Finally, processes may or may not have identities. Processes that do not have identities are *anonymous*, and must all share the same code, with the behavior of two processes diverging only in response to different inputs, different coin-flip outcomes, or different values read from memory.

1.2 Concurrency bounds

We use the terminology of [20]. *Unbounded concurrency* means that there is no limit to the number of processes that may be live at any one time. *Bounded concurrency* holds when there is such a limit in any particular execution. Still more restrictive is *b-bounded concurrency*, where this limit is a fixed value b in all executions.

If the number of processes that are ever live is at most n, we have *n arrivals*. Algorithms for the n arrivals model whose per-process work is bounded as a function of n without n being known to the processes are said to be *adaptive*.

1.3 Wait-free consensus

The *consensus* problem is to get a group of processes in a distributed system to agree on a value.

A *consensus protocol* is an algorithm that produces such an agreement. Each process in a consensus protocol has an input from some specified range and must eventually *decide* on some output from the same range. A process may only decide once, and after deciding it does not participate in the protocol further.

Correct consensus protocols must satisfy the following three conditions:

1. *Agreement.* All processes that decide choose the same value.

2. *Termination.* All non-faulty processes eventually decide.

3. *Validity.* The common decision value is the input of at least one process.

The special case where the input range is $\{0, 1\}$ is called *binary consensus*. For processes with identities, *id-consensus* is consensus where each process's input is its identifier. Finally, we will use *infinitary consensus* to refer to the case where the input range is an arbitrary countably infinite set.

The consensus problem has been known to be unsolvable by deterministic algorithms in asynchronous systems with even a single crash failure since the impossibility result of Fischer *et al.* [19], which was proved for a message-passing model and extended to a shared-memory model by Loui and Abu-Amara [23]. However, randomized solutions for the shared-memory model that tolerate the failure of nearly all processes have been known since the pioneering algorithms of Chor *et al.* [16] (for a weak adversary) and Abrahamson [1] (for a strong adversary). Subsequent efforts by many authors [2, 4, 5, 7, 9–13, 15, 17, 25] have steadily reduced the cost of solving consensus with n processes to the current best known bounds of $O(n \log^2 n)$ operations per process for the strong adversary [5] and $O(\log n)$ operations per process for a weak adversary [9].

Most known wait-free consensus algorithms are based on a reduction from consensus to the *weak shared coin* problem, and we, too, use such a reduction in Section 2. A weak shared coin is a protocol that allows processes to combine their individual coin flips (which are not immediately visible to other processes, and thus vulnerable to delay or interception by the adversary) into a common global coin flip, which for each possible value c will appear as c to all processes with at least a constant *defiance probability* [4]. Known shared coin protocols for a strong adversary are all built using some form of random voting, where each process generates many random votes based on its own internal coin, and the global coin value is determined by a (possibly weighted) majority of the local coins. In such protocols, a strong adversary can seize control of the shared coin by selectively killing off processes that are about to write out votes for the wrong value. But if enough votes are generated, the size of the majority is likely to be large enough that removing the lost votes does not change the winner.

With unweighted voting and n processes, the adversary can dispose of up to $n - 1$ votes, and so $\Theta(n^2)$ votes are needed to get at least a constant probability of an unbiased outcome. Obviously any algorithm that depends on performing $\Theta(n^2)$ operations will be difficult when n is infinite, so instead we use a weighted voting scheme in which each process is assigned an identity starting at 1, and processes with larger identities cast smaller votes (this mechanism is described in Section 3.1). Though the adversary can still destroy infinitely many votes, by carefully choosing the weights according to a convergent series we prevent these infinitely many votes from shifting the outcome of the voting by more than a constant amount, which turns out to be enough to construct a shared coin. The cost that we pay for weighting votes is that the expected work done by the i-th process is now a function of i; however, an argument based on the lower bound of [3] shows that we cannot hope to do much better.

For a weak adversary, we can rely much more on the adversary's inability to see votes until after the processes do. This allows us to construct a weak shared coin whose per-process work does not depend on the process's identity, provided we can assume either bounded concurrency or bounded arrivals. These algorithms are described in Section 3.2.

Our reduction to weak shared coin solves only the binary consensus problem. For larger input ranges, we have a reduction from infinitary consensus to binary consensus, described in Section 2.2. Infinitary consensus turns out to be necessary for our universal construction in Section 5.

1.4 Naming

The *naming problem* requires assigning unique identifiers to anonymous processes. It is known that assigning unique identifiers with probability 1 is impossible, even for finitely many processes and even if consensus is available as a subroutine [14]. However, it is still possible to assign unique identifiers with high probability.

We give several algorithms for assigning unique identifiers to anonymous processes in Section 4. These work with constant probability, and can be used to build anonymous shared coins from standard shared coins, assuming the standard shared coin terminates even when processes share an identifier.

1.5 Summary of results

Our results for consensus are summarized in Tables 1 and 2. Table 1 gives upper bounds for consensus; these are all based on the reduction from consensus to weak shared coins given in Section 2.1. Table 2 gives lower bounds.

2. CONSENSUS

Our basic tool is a generic binary consensus protocol based on the weak-adversary consensus protocol of [15]. This protocol works in every variant of the model we consider, provided it is supplied with an appropriate shared coin algorithm as a subroutine. To implement infinitary consensus, we use an unbalanced tournament tree similar to that used by Attiya and Bortnikov for adaptive mutual exclusion [6]. We complement these positive results with some lower bounds on per-process work, based on the lower bound of [3].

2.1 Binary consensus

In the full paper, we show that a simplified version of the multi-writer register consensus protocol from [15] solves consensus for infinite arrivals given any weak shared coin as a subroutine. The algorithm runs in rounds, and terminates after each round with a probability at least equal to the defiance probability of the shared coin for that round.

The round structure permits slowly increasing a "guess" for a bound on the concurrency or number of arrivals. With a shared coin that (a) terminates in time polynomial in this guess, and (b) exhibits a constant defiance probability once the guess is large enough, the consensus algorithm runs with the same expected time as the shared coin except for constant factors. By increasing the guesses using a geometric series, the cost of earlier coins with incorrect guesses increases the total time by only an additional constant factor. This technique is used to get the bounds in Table 1 where the algorithm's performance depends on n or b without actually knowing n or b; it also handles the case of bounded concurrency, where the bound b may vary from execution to execution.

We use special cases of the algorithm for anonymous processes and n arrivals. These are described in detail in the full paper, but we sketch the central ideas here.

With a weak adversary, we run three rounds of the algorithm without a shared coin to enforce validity, and use the preferences at the end of the last round (for undecided processes) as inputs to the $O(\log n)$ weak-adversary consensus protocol of Aumann [9]. The Aumann algorithm is run using identities in the range 1 to n^4 obtained from Algorithm 5, which are unique with probability at least $1 - 1/n^2$. In case the Aumann algorithm fails, its outputs are used as inputs to a second incarnation of the round-based protocol, with no coin in the first three rounds, and the weak-adversary coin of Algorithm 2 in subsequent rounds. This last step may involve $O(b \log b) = O(n \log n)$ work per process if Algorithm 2 actually runs, but the rarity of this event causes the expected cost to disappear in the overall $O(\log n)$ cost.

With anonymous processes, n arrivals, and a strong adversary, we use as our shared coin the $O(n \log^2 n)$ work-per-process coin of Aspnes and Waarts [5], with identities in the range 1 to n^2 again obtained by running Algorithm 5. The most expensive part of the Aspnes-Waarts coin consists of $O(\log^2 n)$ collect operations, in each of which the value of a register for each participating process is obtained. To avoid doing n^2 reads for each of these collects, we use a tree-based data structure for doing efficient *sparse collects*, where only n positions in an array of n^2 registers are nonempty. This data structure (described in the full paper) allows sparse collects to be done with $O(n)$ work per collect, plus a cumulative $O(n \log n)$ overhead that is shared between all collects done by any one process.

For 1-bounded concurrency, we can skip the round-based algorithm entirely, and use an algorithm where each process writes its input to a single common register after first checking to see that no value has previously been written. As we argue at slightly greater length in the full paper, this almost-trivial algorithm solves consensus in 2 operations per process without using randomization, creating a curious situation. While consensus is solvable quickly, building a weak shared coin in this model is arbitrarily expensive, as an examination of the lower bound on weak shared coins from [3] shows that it requires only 1-bounded concurrency.

Finally, we observe that the algorithm interleaving technique of Attiya *et al.* [8] can be used to combine shared coins to get, for example, a consensus protocol for n arrivals in which process i performs $O(\min(T_c \cdot i^{1+\epsilon}, n \log^2 n))$ work, where T_c is the cost of a collect.

2.2 Infinitary consensus

In the full paper, we give an algorithm for reducing infinitary consensus to binary consensus using an unbalanced tournament tree. Each process starts at a leaf of the tree determined by the process's input, and climbs to the root by executing a binary consensus protocol at each node, adopting as its preference at each stage the winner of the pre-

Adversary	Concurrency	Anonymity	Operations/process	Shared coin
strong	unbounded	no	$O(T_c \cdot i^{1+\epsilon})$	Alg. 1
		yes	?	—
	n arrivals	yes	$O(n \log^2 n)$	Alg. 5+ [5]
	bounded	no	$O(T_c \cdot i^{1+\epsilon})$	Alg. 1
		yes	unbounded	Alg. 3 + Alg. 1
	b-bounded, $b \geq 2$	no	$O(T_c \cdot i^{1+\epsilon})$	Alg. 1
		yes	unbounded	Alg. 3 + Alg. 1
	1-bounded	yes	$O(1)$	Not required
weak	n arrivals	yes	$O(\log n)$	Alg. 5+ [9]
	bounded	yes	$O(b \log b)$	Alg. 2
	b-bounded	yes	$O(b \log b)$	Alg. 2

Table 1: Consensus algorithms. Operations/process are worst-case expected, where T_c is the number of operations for one collect. The value of n or b in a concurrency bound is *not* known to the processes. For bounded concurrency, b is the per-execution bound.

Adversary	Concurrency	Anonymity	Operations/process	Proved in
strong	unbounded	no	$\Omega(i/\log^2 i)$	Theorem 1
	n arrivals	no	$\Omega(n/\log^2 n)$	[3]
	bounded	no	?	
	b-bounded, $b \geq 2$	no	?	
	1-bounded	no	$\Omega(1)$	trivial
weak	any	?	?	

Table 2: Lower bounds. Bounded-arrivals and bounded-concurrency lower bounds apply even when n or b is known.

vious consensus protocol. The tree is the same as used in the adaptive mutual exclusion protocol of Attiya and Bortnikov [6]; it has the property that a process with input k must climb through $2\lfloor log\ k\rfloor + 1$ internal nodes (and thus complete $O(\log k)$ binary consensus protocols) to reach the root.

2.3 Lower bounds

Theorem 1. *For any non-anonymous binary consensus protocol with unbounded concurrency and a strong adversary, the processes can be divided into two sets S and T such that: (a) there is an ordering $p_1, p_2, \ldots$ of all processes in S for which the worst-case expected number of write operations performed by p_i is $\Omega(i/\log^2 i)$; and (b) the worst-case expected number of write operations performed by any process in T is unbounded.*

Proof. Let $W(p)$ be the worst-case expected number of writes performed by process p, or ∞ if this quantity is unbounded. Choose an ordering $<$ of the processes by increasing $W(p)$ and let $S = p_1, p_2, \ldots$ be the prefix of this ordering whose order type is that of the natural numbers. Let T consist of all processes not in S.

Because we are ordering by increasing W, we have $W(p_i) \leq W(p_j)$ when $i \leq j$. So for any adversary strategy, the expected total number of writes performed by $p_1, \ldots p_n$ is at most $\sum_{i=1}^{n} W(i) \leq nW(n)$. From [3], there is an adversary strategy that runs only the first n processes and obtains expected $\Omega(n^2/\log^2 n)$ writes; it follows that $W(n) = \Omega(n/\log^2 n)$.

This establishes the lower bound for each process in S. For processes in T, observe that any such process p has $W(p) \geq W(p_n) = \Omega(n/\log^2 n)$ for arbitrarily large n. It follows that $W(p)$ is infinite and that the expected running time of p is unbounded. □

3. SHARED COINS

A *shared coin* with *defiance probability* δ is a protocol that guarantees that with probability at least δ all participants decide 0, and that with probability at least δ all participants decide 1. A *strong shared coin* has $\delta = \frac{1}{2}$. Wait-free strong shared coins are known to be impossible in a model where timing is controlled by a strong adversary [4]. A *weak shared coin* is a shared coin where $0 < \delta < \frac{1}{2}$. In this section, we will show how to construct wait-free weak shared coins under a variety of assumptions about scheduling.

3.1 Convergent voting

The *convergent voting* algorithm (Algorithm 1) implements a weak shared coin for infinitely many processes with identities $1, 2, 3, \ldots$. The algorithm resembles the Aspnes-Waarts shared coin [5], which uses weighted voting to ob-

```
procedure SharedCoin () returns boolean
shared data: array of single-writer registers $r[j]$ for each $j = 1, 2, \ldots$, with fields
             $r[j]$.variance and $r[j]$.vote, both initially 0
local data: temporary weights $w$ and $v$
begin
    repeat
1       w ← $\max\left(\frac{1}{\zeta(1+\epsilon)i^{1+\epsilon}}, \frac{1}{U}\, r[i].\text{variance}\right)$
2       if LocalCoin () = true then
            $v \leftarrow +w$
        else
            $v \leftarrow -w$
        end
3       write $(r[i].\text{variance}, r[i].\text{vote}) \leftarrow (r[i].\text{variance} + w^2, r[i].\text{vote} + v)$
4       Collect $(r[j])$
5       total ← $\sum_j r[j]$.variance
    until total $\geq U$
6   Collect $(r[j])$
7   return $\left(\sum_j r[j].\text{vote} > 0\right)$
end
```

Algorithm 1: Convergent voting: a weak shared coin for a strong adversary. Code for process i.

tain early termination with few processes. Here, we are using weighted voting primarily to allow arbitrarily many processes to participate in the protocol while still limiting the adversary's control, although, as in the Aspnes-Waarts algorithm, we also increase the weight of long-running processes' votes to reduce their worst-case running times.

The overall structure of the algorithm is typical of shared coins based on randomized voting. The processes collectively generate a set of random votes whose sum has large variance, while making sure that the sum of all outstanding votes not yet written to the registers is small. Applying a variant of the Central Limit Theorem (Corollary 9) to the infinite sequence of all votes ever generated shows that the total vote is far from the origin with constant probability; having a small total outstanding vote then ensures that the adversary's ability to hide votes does not change the apparent majority from the correct one. To guarantee that the total outstanding vote is small even with arbitrarily many unwritten votes, we set the size of process i's initial votes using the i-th term in a series that converges to 1, and only allow the process to generate larger votes when it has already contributed more than its fair share of the total variance.

Pseudocode for the convergent voting algorithm is given as Algorithm 1. In Line 1, ζ is the Riemann zeta function given by $\zeta(s) = \sum_{i=1}^{\infty} i^{-s}$. The parameter ϵ, which can be any value greater than 0, controls the asymptotic running time of each process as a function of its identity; choosing different values of ϵ trades off the running times of processes with low identities for those with high identities. The variance bound U, used in the termination test, is a constant that does not depend on i or ϵ, and is chosen so that the size of the majority when a process leaves the main loop is likely to be large. The `LocalCoin` subroutine called in Line 2 is the local coin of process i; the `Collect` subroutine in Lines 4 and 6 is any implementation of an infinite-process collect, for example using the techniques of [20].

Theorem 2. *Let $U \geq 1$. Then process i completes Algorithm 1 with at most $O\left(\zeta(1+\epsilon)i^{1+\epsilon}U\right)$ calls to `Collect` and a similar number of write operations.*

Proof. Fix i. Let w_j be the weight of process i's j-th vote, that is, the value of w it computes in its j-th pass through Line 1. Let $u_j = \sum_{k=1}^{j} w_k^2$; note that this is precisely the value of $r[i]$.variance used to compute w_{j+1}. Since the total weight process i computes in Line 5 includes the squares of the weights of its own votes, process i must leave the loop no later than the first pass j at which u_j exceeds U. We can bound the number of passes it takes through the loop by showing that u_j increases rapidly.

Let $m = \frac{1}{\zeta(1+\epsilon)i^{1+\epsilon}}$ be the value of the first term in Line 1. For small values of j, $w_j = m$ and $u_j = jm^2$. This case continues until u_j first exceeds mU, i.e., until j equals $t_1 = \lceil m^{-1}U \rceil$.

From time t_1 on, we have a new process in which the $\left(\frac{1}{U}\, r[i].\text{variance}\right)$ term dominates. Writing this term as $u_j U^{-1}$, we have

$$u_{j+1} = u_j + u_j^2 U^{-2}, \tag{1}$$

for $j \geq t_1$.

Equation (1) is a difference equation, and it is tempting to try to approximate the progress of u after t_1 with the similar differential equation $u'_t = u_t^2 U^{-2}$, whose solution is $u_t = \frac{1}{c - tU^{-2}}$ for some constant c.

Unfortunately, this approach overestimates the rate of growth of u_j, as it assumes an increasing derivative between j and $j+1$. Instead, we will proceed by showing that u doubles quickly, and sum a series of doubling times to bound the first time at which u exceeds U.

Let t_2 be the first time at which u_{t_2} exceeds $2u_{t_1}$; since $u_j \geq u_{t_1}$ for all $j \geq t_1$, we have

$$t_2 \leq t_1 + \left\lceil \frac{2u_{t_1} - u_{t_1}}{u_{t_1}^2 U^{-2}} \right\rceil = t_1 + \left\lceil u_{t_1}^{-1} U^2 \right\rceil.$$

Letting t_k be the first time at which u_{t_k} first exceeds $2^k u_{t_1}$, a similar argument shows that

$$t_k \leq t_{k-1} + \left\lceil 2^{-(k-1)} u_{t_1}^{-1} U^2 \right\rceil,$$

from which it follows that

$$t_k \leq t_1 + \sum_{\ell=1}^{k} \left\lceil 2^{-(\ell-1)} u_{t_1}^{-1} U^2 \right\rceil \leq t_1 + 2u_{t_1}^{-1} U^2 + k. \tag{2}$$

For $k = \lceil -\lg m \rceil$, $u_{t_k} \geq 2^{-\lg m} u_{t_1} \geq m^{-1} mU = U$. From (2) we then have

$$\begin{aligned} t_k &\leq t_1 + 2(mU)^{-1}U^2 + \lceil -\lg m \rceil \\ &\leq m^{-1}U + 2m^{-1}U - \lg m + 2 \\ &\leq 3\zeta(1+\epsilon)i^{1+\epsilon}U + \lg(\zeta(1+\epsilon)) + (1+\epsilon)\lg i + 2 \\ &= O\left(\zeta(1+\epsilon)i^{1+\epsilon}U\right). \end{aligned}$$

□

Lemma 3. *Fix some execution of Algorithm 1. For each i, let m_i be the maximum weight w computed during this execution in Line 1 by process i. Then*

$$\sum_{i=i}^{\infty} m_i \leq 2, \tag{3}$$

and, for all i,

$$m_i \leq 1. \tag{4}$$

Proof. Since the weights computed in Line 1 can only increase over time, we can take m_i to be the *last* weight computed by process i.

Classify each weight m_i depending on whether it was computed using $\frac{1}{\zeta(1+\epsilon)i^{1+\epsilon}}$ or $\frac{1}{U}\, r[i]$.variance. We will bound the sum of the weights in these two classes separately.

The sum of all weights in the first class is at most

$$\sum_{i=1}^{\infty} \frac{1}{\zeta(1+\epsilon)i^{1+\epsilon}} = \frac{1}{\zeta(1+\epsilon)} \sum_{i=1}^{\infty} \frac{1}{i^{1+\epsilon}} = 1.$$

Now consider some process i whose last weight is given by $\frac{1}{U}\, r[i]$.variance. Since $\frac{1}{U}\, r[i]$.variance $= 0$ during the first pass through the loop, in order for m_i to equal $\frac{1}{U}\, r[i]$.variance, process i must have executed the call to `Collect` in Line 4 and avoided terminating the loop before computing m_i. It follows that this call to `Collect` started at or before the last time τ at which the sum of the variance fields in the registers was less than U, and that at time τ, the value of $r[i]$.variance used to compute m_i had already been written to $r[i]$ in the immediately preceding execution of Line 3. Since this argument applies to all such processes i, the sum of all of their last $r[i]$.variance values cannot exceed the sum of the variance fields in the registers at time τ, and we have

$$\sum_{i=1}^{\infty} \frac{1}{U}\, r[i].\text{variance} \leq \frac{1}{U}\, U = 1.$$

For each i, m_i appears on the left-hand side of one or the other of the two inequalities; it follows that each individual m_i is at most 1, proving (4). Similarly, summing the two inequalities gives (3). □

Theorem 4. *For each $\delta < \frac{1}{2}$, there exists a variance bound U depending only on δ such that Algorithm 1 implements a shared coin with defiance probability δ for any choice of $\epsilon > 0$.*

Proof. Fix some adversary strategy; then each execution of Algorithm 1 is determined by the sequence of local coin-flip values. We will record these using random variables Y_i, where each Y_i is the value $\pm w$ assigned to v as the result of the i-th local coin-flip performed by any of the processes executing Line 2. The Y_i thus track the sequence of all votes and their weights as they are generated, regardless of when (or whether) these votes are eventually written to memory. If only some finite number ℓ votes are generated during some execution, then we set $Y_i = 0$ when $i > \ell$.

Let $\mathcal{F}_i$ be the σ-algebra generated by $Y_1, Y_2, \ldots Y_i$. Thus Y_i is trivially measurable $\mathcal{F}_i$. Furthermore, $\mathrm{E}[Y_i|\mathcal{F}_{i-1}] = 0$, since either $Y_i = 0$ because fewer than i votes were generated, or $Y_i = \pm w$ with equal probability for some weight w that is measurable $\mathcal{F}_{i-1}$. So the Y_i form a martingale difference sequence. We will use Corollary 9 to show that the sum of this sequence exists and has a distribution close to normal. We need two facts to apply Corollary 9: tight bounds on the expected total conditional variance of the Y_i, and an upper bound on the sum of their fourth moments.

First let us characterize the total conditional variance of the Y_i. For each Y_i, the conditional variance $\mathrm{E}[Y_i^2|\mathcal{F}_{i-1}] = Y_i^2 = w^2$ where w is the weight of the i-th generated vote. So $\sum_i \mathrm{E}[Y_i^2|\mathcal{F}_{i-1}] = \sum_i Y_i^2$. We can bound this sum from below by observing that unless $\sum_i Y_i^2$ is at least U, the algorithm does not terminate for any process, which contradicts Theorem 2 and the assumption that at least one process takes infinitely many steps. To get an upper bound, divide the votes Y_i into those that are generated while $\sum_j r[j]$.variance is less than U and those that are generated afterwards. Since each process checks $\sum_j r[j]$.variance in between each pair of consecutive votes, each process can generate at most one "late" vote after this total variance reaches U. From Lemma 3, these late votes have total weight at most 2 and individual weights of at most 1; it follows that the sum of the squares of their weights is at most 2. So we have $U \leq \sum_i Y_i^2 \leq U + 2$, and thus $U \leq \sum_i \mathrm{E}[Y_i^2|\mathcal{F}_{i-1}] \leq U + 2$ as well.

Now let us consider the fourth moments. We have just shown that $\sum_i Y_i^2 \leq U + 2$, and we have that each $Y_i \leq 1$ from Lemma 3. A simple convexity argument shows that the maximum for $\sum_i Y_i^4$ is reached when $U + 2$ of the Y_i are 1 and the rest 0; so $\sum_i Y_i^4 \leq U + 2$ in any execution. Then $\sum_i \mathrm{E}[Y_i^4] = \mathrm{E}\left[\sum_i Y_i^4\right] \leq U + 2$.

Normalize the sequence Y_i by setting $X_i = Y_i U^{-\frac{1}{2}}$; $\{X_i, \mathcal{F}_i\}$ is clearly a martingale difference sequence. Let

$$V_\infty^2 = \sum_{j=1}^{\infty} \mathrm{E}[X_j^2|\mathcal{F}_{i-1}] = \sum_{j=1}^{\infty} \mathrm{E}[Y_j^2 U^{-1}|\mathcal{F}_{i-1}] = U^{-1} \sum_{j=1}^{\infty} Y_j^2.$$

From the preceding discussion we have that this last sum lies between U and $U+2$; thus we have $1 \leq V_\infty^2 \leq 1 + 2U^{-1}$, and so $\mathrm{E}[(V_\infty^2 - 1)^2] \leq 4U^{-2}$. Similarly, compute

$$\sum_{j=1}^{\infty} \mathrm{E}[X_j^4] = \sum_{j=1}^{\infty} \mathrm{E}[Y_j^4 U^{-2}] \leq U^{-2}(U+2) = U^{-1} + 2U^{-2}.$$

Let

$$L_\infty = \mathrm{E}\left[(V_\infty^2 - 1)^2\right] + \sum_{j=1}^{\infty} \mathrm{E}[X_j^4] \leq U^{-1} + 6U^{-2}.$$

Let us insist that U is at least 1, so that we can simplify this bound to $7U^{-1}$.

From Corollary 9, there is a universal constant A such that when $L \leq 1$,

$$\left| \Pr\left[\sum_{j=1}^{\infty} X_i \leq x\right] - \Phi(x) \right| \leq AL_\infty^{1/5} \leq A7^{1/5}U^{-1/5}. \tag{5}$$

Returning to the un-normalized votes Y_i, (5) becomes

$$\left| \Pr\left[\sum_{j=1}^{\infty} Y_i \leq x\right] - \Phi(U^{-1/2}x) \right| \leq AL_\infty^{1/5} \leq A7^{1/5}U^{-1/5}. \tag{6}$$

For any fixed x, in the limit as U goes to infinity, the Φ term goes to $1/2$ and the right-hand side goes to 0. So for sufficiently large U, the probability that the sum of the generated votes exceeds x becomes arbitrarily close to $1/2$.

Of course, no process necessarily sees the total generated vote; instead, each process P sees the total generated vote less some set of votes that were not written out to the registers by the time P did its final collect in Line 6. But P starts this collect only after the sum of the variance fields has reached U, so any vote that P misses must be written after this time. If Q writes a missed vote, it reads a total

```
procedure SharedCoin () returns boolean
shared data: coins$[1 \dots 2b \lceil \lg b \rceil + 1]$, an array holding
             values from the set $\{0, 1, \perp\}$, all initial-
             ized to $\perp$.
local data: count$[0]$, count$[1]$, initially 0.
begin
    for $i \leftarrow 1$ to $2b \lceil \lg b \rceil + 1$ do
        if coins$[i] \neq \perp$ then
            coins$[i] \leftarrow$ LocalCoin()
        end
    end
1   for $i \leftarrow 1$ to $2b \lceil \lg b \rceil + 1$ do
2       $c \leftarrow$ coins$[i]$
3       count$[c]$ = count$[c] + 1$
    end
    if count$[0] >$ count$[1]$ then
        return 0
    else
        return 1
    end
end
```

Algorithm 2: Row of coins: a weak shared coin for b-bounded concurrency and a weak adversary.

variance greater than U during its collect in Line 4, and generates no more votes. It follows that the votes not seen by P include at most one vote per process, and that the sum of all of these missing votes shifts the total by at most 2 by Lemma 3. Thus if the total generated vote exceeds 2, then all processes return true; similarly, if the total generated vote is less than -2, all processes return false. Substituting ± 2 for x shows that the probability of either of these events, which gives a lower bound on the defiance probability δ, can be set arbitrarily close to $1/2$ for sufficiently large U. □

3.2 Weak-adversary coin

Pseudocode for a simple weak shared coin algorithm for a weak adversary and b-bounded concurrency is given in Algorithm 2. The key to the correctness of the algorithm is that when the array is first filled, there are at most $b - 1$ outstanding unwritten votes, any process sees one of at most b^2 distinct subsets of these votes, and that these extra votes modify the $2b \lceil \lg b \rceil + 1$ common votes in each view according to a biased random walk whose extent we can bound with Azuma's inequality.

Theorem 5. *In every execution, Algorithm 2 terminates in $O(b \log b)$ steps. Given a weak adversary and b-bounded concurrency, it implements a weak shared coin with a constant defiance probability.*

Proof. Let an invocation's *view* of coins be the sequence of values it reads from coins during its final collect starting in Line 1. We will begin by showing that all infinitely many invocations of SharedCoin between them observe at most b^2 distinct views of coins; we will then show that all of these views have the same majority value with at least a constant probability.

Let C_0 be the state of coins following the first write W to coins$[2b \lceil \lg b \rceil + 1]$; let t_0 be the time of this first write. Any invocation of SharedCoin that starts after this write will see only non-$\perp$ entries in coins and will perform no writes. Each pending invocation at time t_0 will perform at most one write after t_0. There can be at most b invocations spanning t_0, and one of them executes W, so at most $b - 1$ writes occur after t_0. Let $t_1, t_2, \dots t_k$, where $k \leq b - 1$, be the times of these writes and C_i be the state of coins after the write at t_i.

Now consider the values read from coins by some process P during its final collect. Note that this collect cannot start before t_0, since before its final collect, P must have observed or written a non-$\perp$ entry in every position in coins. There are two cases: (1) If P's final collect starts and ends between two times t_i and t_{i+1}, then P observes C_i; this accounts for at most b distinct views of coins. (2) If P's final collect spans one or more of the times t_i, the view it obtains may or may not include the values written at these times. However, there can be at most b invocations spanning each of the at most $b - 1$ writes; even if these invocations are all distinct, between them they see at most $b(b - 1)$ different views of coins.

Adding the two cases together gives at most b^2 distinct views.

Now let us show that these b^2 views all yield the same majority with constant probability. Let X_0 be the number of 1 votes in coins at time t_0. By the normal approximation to the binomial distribution, for large enough b there is a constant probability that X_0 exceeds $b \lceil \lg b \rceil + 5\sqrt{2b \lceil \lg b \rceil + 1}$. We will show that when this event occurs, all views return a 1 majority with constant probability. The 0 case is symmetric.

Let X_i be the the number of 1 votes read as part of the i-th view, where each view is ordered by the time at which the first invocation that obtains that view finishes its final collect. The difference between X_0 and X_i is determined by the values of up to $b - 1$ coin-flips written out after t_0, and each view can be specified by identifying which subset of these $b - 1$ coin-flips it includes. We will show that with high probability this difference is less than $5\sqrt{2b \lceil \lg b \rceil + 1}$, and thus not enough to change the majority value in the view from that in state C_0.

Let $N = 2b \lceil \lg b \rceil + 1$ be the size of coins. Conditioning on X_0, the probability that any given coin in C_0 is a 1 is $\frac{X_0}{N}$. Replacing a coin in C_0 with a new coin-flip thus changes the total number of ones by $\frac{1}{2} - \frac{X_0}{N}$ on average, and we can represent the increment due the j-th coin that is replaced in view i as $\frac{1}{2} - \frac{X_0}{N} + Y_{ij}$, where Y_{ij} is a zero-mean random variable with $|Y_{ij}| \leq 2$.

Continuing to condition only on X_0, the difference $X_i - X_0$ is given by $\frac{m}{2} - \frac{mX_0}{N} + \sum_{j=1}^{m} Y_{ij}$, $m \leq b - 1$ is the number of coins replaced in view i and Y_{ij} is as above. The first two terms are at least $-b + 1$. For the last term, Hoeffding's inequality implies that

$$\begin{aligned}
\Pr\left[\sum_{j=1}^{m} Y_{ij} \leq -4\sqrt{2b \lceil \lg b \rceil + 1}\right] & \\
\leq \quad & \exp\left(-\frac{\left(4\sqrt{2b \lceil \lg b \rceil + 1}\right)^2}{8m}\right) \\
\leq \quad & \exp\left(-2\frac{2b \lceil \lg b \rceil + 1}{b}\right) \\
\leq \quad & \exp(-4 \log b) = b^{-4}.
\end{aligned}$$

It follows that, if $X_0 \geq b\lceil \lg b \rceil + 5\sqrt{2b\lceil \lg b \rceil + 1}$, then $X_i > b\lceil \lg b \rceil$ with probability at least b^{-4}.

In this analysis, we have neglected the complication that the adversary may be able to choose which new coins to include in view X_i after observing the effects of earlier views. However, the adversary's ability to observe these earlier views is limited. The coins read in Line 2 and counted in Line 3 affect only the internal state of the process, which is assumed not to be observable by the adversary, and even the counts stored in `count[]` are discarded when the call to `SharedCoin` returns. So the only event observable by the adversary is the return value (as this may affect the later behavior of the calling process), which tells the adversary only whether its previous views successfully produced a conflicting majority. We will use this fact to show that the adversary's ability to choose which coins to include does not give it enough power in most cases to change the outcome of the shared coin.

Let A_i be the event that X_i has a majority of zeroes. Note that the coins included in X_i may depend on the outcome of earlier events A_j. Let B_k be the event $\bigvee_{i=1}^{k} A_i$, We will show by induction that when $X_0 \geq b\lceil \lg b \rceil + 5\sqrt{2b\lceil \lg b \rceil + 1}$, $\Pr[B_k|X_0] \leq kb^{-4}$ for each k. Clearly the base case holds when $k = 0$.

Now suppose that it holds for $k - 1$. Let us compute the probability that A_k occurs, conditioned on B_{k-1} not occurring (that is, on the adversary's not having won yet). For any possible view X_k, we have that the probability that X_k has a zero majority is at most b^{-4} without conditioning on $\neg B_{k-1}$. So its probability conditioned on $\neg B_{k-1}$ is at most $\frac{b^{-4}}{1-\Pr[B_{k-1}]}$. As this bound holds for *any* view, it holds in particular for whichever view is chosen to be the k-th view by the adversary.

Now we can compute

$$\begin{aligned}
&\Pr[B_k|X_0] = \\
&\quad \Pr[B_{k-1}|X_0] + \Pr[\neg B_{k-1}|X_0]\Pr[A_k|\neg B_{k-1}, X_0] \\
&\leq \Pr[B_{k-1}|X_0] + (1 - \Pr[B_{k-1}|X_0])\frac{b^{-4}}{1 - \Pr[B_{k-1}|X_0]} \\
&= \Pr[B_{k-1}|X_0] + b^{-4} \\
&\leq (k-1)b^{-4} + b^{-4} \\
&= kb^{-4}.
\end{aligned}$$

With $k \leq b^2$ views, we have a probability of at most b^{-2} that at least one of them has a zero majority, and thus a probability of at least $1 - b^{-2}$ that all have a one majority. Multiplying by the constant probability that $X_0 \geq b\lceil \lg b \rceil + 5\sqrt{2b\lceil \lg b \rceil + 1}$ then gives a constant probability that all invocations return 1. □

3.3 Lower bounds

In Algorithm 1, the assumption of process identities is vital. In fact, we can show that with anonymous processes it is not possible to construct a weak shared coin in the same model. We conjecture that a similar result holds for consensus.

THEOREM 6. *There is no anonymous weak shared coin algorithm for unbounded concurrency and a strong adversary.*

The intuition behind the proof is as follows. If some process P returns a particular value when run in isolation with some particular low-probability set of coin-flips, the adversary can simulate this process P with a vast army of anonymous clones. So long as each clone continues to flip coins the same way that P did, the adversary will continue to run it in lockstep with the other clones; in this way no clone ever observes that it is not alone. With a large enough set of clones, the adversary can amplify the probability that some clone flips all coins the right way arbitrarily, thus turning low probability outcomes of the global coin into high probability ones.

```
procedure GetID_b () returns integer
shared data: integer Id, initially 0
begin
    $k \leftarrow \mathsf{Id} + 1$
    $\mathsf{Id} \leftarrow k$
    Let $r$ be chosen uniformly from $\{1 \ldots Nb^{3k}\}$
    return $Nkb^{3k} + r$
end
```

Algorithm 3: Assigning unique identifiers with b-bounded concurrency.

4. NAMING

We reduce the problem of building a weak shared coin for anonymous processes to the problem for processes with identities by using a randomized algorithm that assigns a unique identifier to every process with high probability. We give three algorithms for assigning identifiers. The first two work under the assumption of b-bounded concurrency, where b is known, and the third under the assumption of at most n arrivals, where n is known. The assumption that b or n is known can be removed at a higher level; see Section 2.1. The assumption that some bound b or n exists in each execution is necessary given a strong adversary, using an argument similar to the proof of Theorem 6.

Algorithm 3 assigns unique identifiers with constant probability even with infinite arrivals, provided concurrency is bounded by a known constant b. The essential idea is that preliminary identifiers obtained by reading a single multi-writer register Id divide the processes into finite classes, where at most b^k processes write k to Id. Within each class, we can prevent duplicate identities by choosing identifiers randomly from a large range, the size of this range determined by b, the preliminary identifier obtained by a process, and a parameter N.

THEOREM 7. *With b-bounded concurrency, Algorithm 3 returns duplicate values with probability at most $\frac{1}{N}$.*

PROOF. First let us show that Id is assigned the value k in at most b^k executions of `GetID`$_b$. Consider some execution consisting of infinitely many invocations of `GetID`$_b$. Let us build a tree of write operations to Id; we will formally set the root of this tree to be a hypothetical pre-execution write of the 0 initially found in Id, and each write operation that writes a value $k > 0$ in some invocation C will have as its parent the write operation that previously wrote the value $k - 1$ read from Id by C.

Observe that with b-bounded concurrency, each node W in the tree has at most b children. All such children must be writes of processes that (a) read Id after W but before the

```
procedure LazyClock () returns integer
shared data: integer $\mathsf{clock}[i]$ for $i = 0, 1, \ldots b-1$, initially 0.
begin
    $c \leftarrow \max(\mathsf{clock}[0], \mathsf{clock}[1], \ldots \mathsf{clock}[b-1]) + 1$
    $\mathsf{clock}[c \bmod b] \leftarrow c$
    return $c$
end
```

Algorithm 4: Lazy clock for b-bounded concurrency.

next write to Id, and (b) write Id no sooner than the first write W' after W; thus, all the processes are simultaneously alive immediately before W', and their number is at most the concurrency bound b. Thus the number of nodes at depth k is at most b^k, and since all writes of k are at depth k, at most b^k writes of k occur.

Now let us estimate the number of collisions between identifiers for invocations that choose the same value k. (The range of r in Line 3 and the return value formula in Line 4 are carefully chosen so that there can be no collisions between invocations that choose different k). If $b = 1$, the number is 0, as there is a unique invocation in each class. Otherwise, there are at most $\binom{b^k}{2} < b^{2k}/2$ pairs of invocations in this class, and the probability that any two such invocations return the same value is exactly b^{-3k}/N. Summing over all pairs gives an expected number of collisions between invocations with preliminary identifier k less than $b^{-k}/(2N)$. Summing over all k gives $\sum_{k=1}^{\infty} \frac{1}{2N} b^{-k} = \frac{1}{2N(1-1/b)} \leq \frac{1}{N}$ (when $b > 1$). □

Algorithm 3 is adequate for our purposes, as we must pay arbitrarily large worst-case costs with unbounded arrivals, and getting hideously large identifiers cannot worsen this already poor bound. However, we can imagine that for other applications it might be useful to make stronger guarantees about the size of identifiers. Algorithm 4 is designed to substitute for the single register in Algorithm 3; it guarantees that at most b^2 invocations return the same value, and has additional pleasant counter-like properties. In particular, we can use Algorithm 4 to guarantee that the i-th process to complete the LazyClock procedure obtains an identity that is polynomial in i. We discuss this algorithm further in the full paper.

With bounded arrivals, we can just choose random identifiers from a sufficiently large range. This has the advantage of allowing us to guarantee that the identifiers are (relatively) small. For symmetry, we give the algorithm for doing this as Algorithm 5. It gives a probability of collision of at most $1/N$.

```
procedure GetID$_n$ () returns integer
begin
    Let $r$ be chosen uniformly from $\{1 \ldots N\binom{n}{2}\}$.
    return $r$
end
```

Algorithm 5: Assigning unique identifiers with n arrivals.

5. A UNIVERSAL CONSTRUCTION

Herlihy [22] showed that consensus is *universal* in the sense that it permits a wait-free implementation of any linearizable shared object with total operations given its sequential specification. The essence of the construction is that each process that wishes to perform an operation announces the operation, performs a collect of all other pending operations of other, possibly slower processes, and then runs consensus to get agreement on which sequence of these operations will actually occur. The process repeats this cycle until its pending operation is included in one of the agreed-upon sequences. Using the infinite-arrivals collect of [20] and the infinitary consensus protocol of Section 2.2, essentially the same construction continues to work in the infinite-arrivals model.

6. OPEN PROBLEMS

Several open problems are apparent from the missing entries in Tables 1 and 2. The most salient is the question of whether it is possible to build an anonymous consensus protocol for unbounded concurrency and a strong adversary. The usual reduction to shared coin does not work, because the adversary can make the defiance probability of a shared coin arbitrarily small (Theorem 6). We suspect that an FLP-style argument similar to that of [3] should work in this case, but technical complications arise that leave open the possibility of a consensus protocol that runs for an arbitrarily large but finite time, and that does not embed a weak shared coin.

Our bounds for the weak adversary are not as tight as for the strong adversary; in particular, we suspect that the per-process work for weak-adversary consensus with bounded concurrency could be much improved.

7. REFERENCES

[1] Karl R. Abrahamson. On achieving consensus using a shared memory. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 291–302, Toronto, Ontario, Canada, August 1988.

[2] James Aspnes. Time- and space-efficient randomized consensus. *Journal of Algorithms*, 14(3):414–431, May 1993.

[3] James Aspnes. Lower bounds for distributed coin-flipping and randomized consensus. *Journal of the ACM*, 45(3):415–450, May 1998.

[4] James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, September 1990.

[5] James Aspnes and Orli Waarts. Randomized consensus in expected $O(N \log^2 N)$ operations per processor. *SIAM Journal on Computing*, 25(5):1024–1044, October 1996.

[6] Hagit Attiya and Vita Bortnikov. Adaptive and efficient mutual exclusion. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 91–100, Portland, Oregon, July 2000.

[7] Hagit Attiya, Danny Dolev, and Nir Shavit. Bounded polynomial randomized consensus. In *Proceedings of the Eighth Annual ACM Symposium on Principles of*

Distributed Computing, pages 281–293, Edmonton, Alberta, Canada, 14–16 August 1989.

[8] Hagit Attiya, Nancy Lynch, and Nir Shavit. Are wait-free algorithms fast? *Journal of the ACM*, 41(4):725–763, July 1994.

[9] Yonatan Aumann. Efficient asynchronous consensus with the weak adversary scheduler. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 209–218, Santa Barbara, California, 21–24 August 1997.

[10] Yonatan Aumann and Michael A. Bender. Efficient asynchronous consensus with the value-oblivious adversary scheduler. In Friedhelm Meyer auf der Heide and Burkhard Monien, editors, *Automata, Languages and Programming, 23rd International Colloquium*, volume 1099 of *Lecture Notes in Computer Science*, pages 622–633, Paderborn, Germany, 8–12 July 1996. Springer-Verlag.

[11] Yonatan Aumann and Avivit Kapah-Levy. Cooperative sharing and asynchronous consensus using single-reader single-writer registers. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 61–70, Baltimore, Maryland, January 1998.

[12] Gabi Bracha and Ophir Rachman. Approximated counters and randomized consensus. Technical Report 662, Technion, 1990.

[13] Gabriel Bracha and Ophir Rachman. Randomized consensus in expected $O(n^2 \log n)$ operations. In Sam Toueg, Paul G. Spirakis, and Lefteris M. Kirousis, editors, *Distributed Algorithms, 5th International Workshop*, volume 579 of *Lecture Notes in Computer Science*, pages 143–150, Delphi, Greece, 7–9 October 1991. Springer, 1992.

[14] Harry Buhrman, Alessandro Panconesi, Riccardo Silvestri, and Paul M. B. Vitanyi. On the importance of having an identity or is consensus really universal? In *International Symposium on Distributed Computing*, pages 134–148, 2000.

[15] Tushar Deepak Chandra. Polylog randomized wait-free consensus. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 166–175, Philadelphia, Pennsylvania, USA, 23–26 May 1996.

[16] Benny Chor, Amos Israeli, and Ming Li. Wait-free consensus using asynchronous hardware. *SIAM Journal on Computing*, 23(4):701–712, August 1994.

[17] Cynthia Dwork, Maurice Herlihy, Serge Plotkin, and Orli Waarts. Time-lapse snapshots. *SIAM Journal on Computing*, 28(5):1848–1874, 1999.

[18] Faith Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *Journal of the ACM*, 45(5):843–862, September 1998.

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

[20] Eli Gafni, Michael Merritt, and Gadi Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, pages 161–169, Newport, Rhode Island, August 2001.

[21] P. Hall and C.C. Heyde. *Martingale Limit Theory and Its Application.* Academic Press, 1980.

[22] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[23] Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In Franco P. Preparata, editor, *Advances in Computing Research*, volume 4. JAI Press, 1987.

[24] M. Pease, R. Shostak, and L. Lamport. Reaching agreements in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[25] Michael Saks, Nir Shavit, and Heather Woll. Optimal time randomized consensus—making resilient algorithms fast in practice. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 351–362, San Francisco, California, 28–30 January 1991.

APPENDIX

Appendix: A Central Limit Theorem

The proof of Theorem 4 uses a version of the Central Limit Theorem that applies to martingales with infinitely many steps. We show that this variant holds in Corollary 9 below. The corollary follows from a theorem of Hall and Heyde [21], part of which we restate here as Lemma 8:

LEMMA 8 ([21]). *Let* $\{S_i = \sum_{j=1}^{i} X_j, \mathcal{F}_i, i = 0, 1, 2, \ldots n\}$ *be a zero-mean martingale. Let*

$$V_n^2 = \sum_{j=1}^{n} \mathrm{E}\left[X_j^2 | \mathcal{F}_{j-1}\right],$$

and

$$L_n = \mathrm{E}\left[(V_n^2 - 1)^2\right] + \sum_{j=1}^{n} \mathrm{E}[X_j^4].$$

There exists a fixed constant A such that, when $L_n \leq 1$,

$$|\Pr[S_n \leq x] - \Phi(x)| \leq A L_n^{1/5}, \tag{7}$$

where Φ *is the normal distribution function.*

The restriction to finite times is inconvenient for us, so we'll get rid of it by taking limits. The result is Corollary 9:

COROLLARY 9. *Let* $\{S_i = \sum_{j=1}^{i} X_j, \mathcal{F}_i, i = 0, 1, 2, \ldots\}$ *be a zero-mean martingale. Let*

$$V_\infty^2 = \sum_{j=1}^{\infty} \mathrm{E}\left[X_j^2 | \mathcal{F}_{j-1}\right],$$

and

$$L_\infty = \mathrm{E}\left[(V_\infty^2 - 1)^2\right] + \sum_{j=1}^{\infty} \mathrm{E}[X_j^4],$$

and suppose that V_∞^2 *is bounded with probability* 1.

There exists a fixed constant A such that, when $L_\infty \leq 1$, $S_\infty = \lim_{n\to\infty} S_n$ *exists and*

$$|\Pr[S_\infty \leq x] - \Phi(x)| \leq A L_\infty^{1/5}. \tag{8}$$