Wait-Free Consensus

James Aspnes

July 24, 1992

CMU-CS-92-164

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

©1992 James Aspnes.

This research was supported by an IBM Graduate Fellowship and an NSF Graduate Fellowship.

School of Computer Science

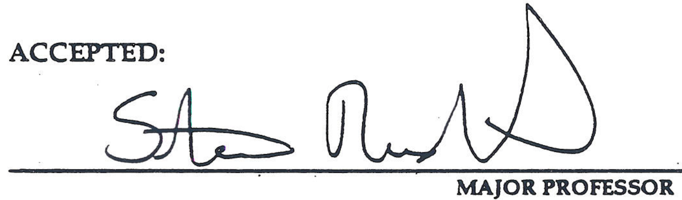
DOCTORAL THESIS
in the field of
Computer Science

Wait-Free Consensus

JAMES ASPNES

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

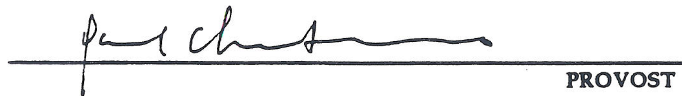
ACCEPTED:



MAJOR PROFESSOR — 7/21/92 **DATE**

DEAN — 8/24/92 **DATE**

APPROVED:



PROVOST — 25 August 1992 **DATE**

Abstract

Consensus is a decision problem in which n processors, each starting with a value not known to the others, must collectively agree on a single value. If the initial values are equal, the processors must agree on that common value; this is the **validity** condition. A consensus protocol is **wait-free** if every processor finishes in a finite number of its own steps regardless of the relative speeds of the other processors, a condition that precludes the use of traditional synchronization techniques such as critical sections, locking, or leader election. Wait-free consensus is fundamental to synchronization without mutual exclusion, as it can be used to construct wait-free implementations of arbitrary concurrent data structures. It is known that no deterministic algorithm for wait-free consensus is possible, although many randomized algorithms have been proposed.

I present two algorithms for solving the wait-free consensus problem in the standard asynchronous shared-memory model. The first is a very simple protocol based on a random walk. The second is a protocol based on weighted voting, in which each processor executes $O(n \log^2 n)$ expected operations. This bound is close to the trivial lower bound of $\Omega(n)$, and it substantially improves on the best previously-known bound of $O(n^2 \log n)$, due to Bracha and Rachman.

Acknowledgments

I would like to begin by thanking the members of my committee. My advisor, Steven Rudich, has always been willing to provide encouragement. Danny Sleator showed me that research is better done as play than as work. Merrick Furst has always been a source of interesting observations and ideas. Maurice Herlihy introduced me to wait-free consensus when I first arrived at Carnegie Mellon; his keen insights into distributed computing have been a continuing influence on my work.

I would like to thank my parents for their warm support and encouragement.

I would like to thank the many other students who lightened the monastic burdens of graduate student life. David Applegate in particular was always ready to supply a new problem or a new toy.

Finally, I would like to thank my beloved wife, Nan Ellman, who waited longer and more patiently for me to finish than anyone.

Contents

1 Introduction **1**

2 The Asynchronous Shared-Memory Model **8**
- 2.1 Basic elements 8
- 2.2 Time and asynchrony 9
- 2.3 Randomization 10
- 2.4 Relation to other models 10
- 2.5 Performance measures 12

3 Consensus and Shared Coins **14**
- 3.1 Consensus 14
- 3.2 Shared coins 16
- 3.3 Consensus using shared coins 17

4 Consensus Using a Random Walk **20**
- 4.1 Random walks 21
- 4.2 The robust shared coin protocol 23
- 4.3 Implementing a bounded counter with atomic registers 29
- 4.4 The randomized consensus protocol 31

5 Consensus Using Weighted Voting **36**
- 5.1 Introduction 36
- 5.2 The shared coin protocol 38
- 5.3 Martingales 40
  - 5.3.1 Knowledge, σ-algebras, and measurability 41
  - 5.3.2 Definition of a martingale 42
  - 5.3.3 Gambling systems 43

List of Figures

3.1 Consensus from a shared coin. 18

4.1 Robust shared coin protocol. 24
4.2 Pictorial representation of robust shared coin protocol. 24
4.3 The protocol as a controlled random walk. 27
4.4 Pseudocode for counter operations. 30
4.5 Consensus protocol. 32
4.6 Counter scan for randomized consensus protocol. 32

5.1 Shared coin protocol. 38

List of Tables

6.1 Comparison of consensus protocols. 61

Chapter 1

Introduction

Consensus [CIL87] is a tool for allowing a group of processors to collectively choose one value from a set of alternatives. It is defined as a decision problem in which n processors, each starting with a value (0 or 1) not known to the others, must collectively agree on a single value. (The restriction to a single bit does not prevent the processors from choosing between more than two possibilities since they can run just run a one-bit consensus protocol multiple times.) The processors communicate by reading from and writing to a collection of registers; each processor finishes the protocol by deciding on a value and halting. A consensus protocol is **wait-free** if each processor makes its decision after a finite number of its *own* steps, regardless of the relative speeds or halting failures of the other processors. In addition, a consensus protocol must satisfy the **validity** condition: if every processor starts with the same input value, every processor decides on that value. This condition excludes trivial protocols such as one where every processor always decides 0.

The asynchronous shared-memory model is an attempt to capture the effect of making the weakest possible assumptions about the timing of events in a distributed system. At each moment an adversary scheduler chooses one of the n processors to run. No guarantees are made about the scheduler's choices— it may start and stop processors at will, based on a total knowledge of the state of the system, including the contents of the registers, the programming of the processors, and even the internal states of the processors. Since the scheduler can always simulate a halting failure by choosing not to run a processor, the model effectively allows up to $n-1$ halting failures. The

problem. Their situation is very much like the situation of two people facing each other in a narrow hallway; neither person has any stake in whether they pass on the left or the right, but if one goes left and the other right they will bump into each other and make no progress. When A and B are deterministic processors under the control of a malicious adversary scheduler, we can show the scheduler will be able to use its knowledge of their state and its control over the timing of events to keep A and B oscillating back and forth forever between the two possible decision values.

Here is what happens. Since each processor is deterministic, at any given point in time it has some **preference**, defined as the value ("left" or "right" in the hallway example) that it will eventually choose if the other processor executes no more operations [AH90a].[1] At the beginning of the protocol, each processor's preference is equal to its input, because without knowing that some other processor has a different input it must cautiously decide on its own input to avoid violating the validity condition. So we can assume that initially processor A prefers to pass on the left, and processor B on the right.

Now the scheduler goes to work. It stops B and runs A by itself. After some finite number of steps, A must make a decision (to go left) and halt, or the termination condition will be violated. But before A can finish, it must make sure that B will make the same decision it makes, or the consistency condition will be violated. So at some point A must tell B something that will cause B to change its preference to "left", and in the shared-memory model this message must take the form of a write operation (since B can't see when A does a read operation). Immediately *before* A carries out this critical write, the scheduler stops A and starts B.

This action puts B in the same situation that A was in. B still prefers to go right, and after some finite number of steps it must tell A to change its preference to "right". When this point is reached either one of two conditions holds: either B has done something to neutralize A's still undelivered demand that B change it preference, in which case the scheduler just stops B and runs A again, or both A and B are about to deliver writes that will cause the other to change its preference. In this case, the scheduler allows both of

[1]This unfortunate possibility is unlikely to occur in the real-world hallway situation, assuming healthy participants, but it is allowed by the asynchronous shared-memory model since the adversary can always choose never to run the other processor again.

idea. Each processor repeatedly adds random ± 1 votes to a common pool until some termination condition is reached. Any processor that sees a positive total vote decides 1, and those that see a negative total vote decide 0. Intuitively, because all of the processors are executing the same loop over and over again, the adversary's power is effectively limited to blocking votes it dislikes by stopping processors in between flipping their local coins to decide on the value of the votes and actually writing the votes out to the registers. The adversary's control is limited by running the protocol for long enough that the sum of these blocked votes is likely to be only a fraction of the total vote, a process that requires accumulating $\Omega(n^2)$ votes.

In the original shared coin protocol of Aspnes and Herlihy [AH90a], each processor decides on a value when it sees a total vote whose absolute value is at least a constant multiple of n from the origin. For each of the expected $\Theta(n^2)$ votes, $\Theta(n^2)$ register operations are executed, giving a total running time of $\Theta(n^4)$ operations. Unfortunately, both the implementation of the counter representing the position of the random walk and the mechanism for repeatedly running the shared coin require a potentially unbounded amount of space. This problem was corrected in a protocol of Attiya, Dolev, and Shavit [ADS89], which retained the multiple rounds of its predecessor but cleverly reused the space used by old shared coins once they were no longer needed.

A simpler descendent of the shared coin protocol of Aspnes and Herlihy, which also requires only bounded space, is the shared coin protocol described in Chapter 4. This protocol, by using a more sophisticated termination condition, guarantees that the processors *always* agree on its outcome. A simple modification of this protocol gives a consensus protocol that does not require multiple executions of a shared coin; which can be implemented using only three $O(\log n)$-bit counters, supporting increment, decrement, and read operations; and which runs in only $\Theta(n^2)$ expected *counter* operations. However, this apparent speed is lost in the implementation of the counter, because $\Theta(n^2)$ register operations are needed for each counter operation, giving it the same running time of $\Theta(n^4)$ expected register operations as its predecessors. Since the consensus protocol of Chapter 4 first appeared [Asp90], other researchers [BR90, DHPW92] have described weaker primitives that act sufficiently like counters to make the protocol work and which use only a linear number of register operations for each counter operation. Using these primitives in place of the counters gives a consensus protocol that runs in

generate $\Omega(n^2)$ local coin flips. The essence of wait-freeness is bounding the work done by a single processor, despite the failures of other processors. But the bound on the work done by a single processor, in every one of these protocols, is asymptotically no better than the bound on the work done by all of the processors together.

Chapter 5 shows that wait-free consensus can be achieved without forcing a fast processor to do most of the work. I describe a shared coin protocol in which the processors cast votes of steadily increasing weights. In effect, a fast processor or a processor running in isolation becomes "impatient" and starts casting large votes to finish the protocol more quickly. This mechanism does grant the adversary greater control, because it can choose from up to n different weights (one for each processor) when determining the weight of the next vote to be cast. One effect of this control is that a more sophisticated analysis is required than for the unweighted-voting protocols. Still, with appropriately-chosen parameters the protocol guarantees that each processor finishes after only $O(n \log^2 n)$ expected operations.

The organization of the dissertation is as follows. Chapters 2 and 3 provide a framework of definitions for the material in the later chapters. Chapter 2 describes the asynchronous shared-memory model in detail and compares it with other models of distributed systems. Chapter 3 formally defines the consensus problem and its relationship to the problem of constructing shared coins. The main results appear in Chapters 4 and 5. Chapter 4 describes the simple consensus protocol based on a random walk. Chapter 5 describes the faster protocol based on weighted voting. Finally, Chapter 6 compares these results to other solutions to the problem of wait-free consensus and discusses possible directions for future work.

Much of the content of Chapters 4 and 5 also appears in [Asp90] and [AW92], respectively. Some of the material in Chapter 3 is derived from [AH90a].

read and write operations act as if they take place instantaneously: they never fail, and the result of concurrent execution of multiple operations on the same register is consistent with their having occurred sequentially.

The assumptions behind atomicity may appear to be rather strong, especially in a model that is designed to be as harsh as possible. However, it turns out that atomic registers are not powerful enough to implement deterministically such simple synchronization primitives as queues or test-and-set bits [Her91], and may be constructed efficiently from much weaker primitives in a variety of ways [BP87, IL87, NW87, Pet83, SAG87]. So in fact the apparent strength of atomic registers is somewhat illusory.

2.2 Time and asynchrony

The systems represented by the model may have many events occurring concurrently. However, because the only communication between processors in the system is by operations on atomic registers, it is possible to represent its behavior using a **global-time model** [BD88, Lam86a, Lam86b]. Instead of treating operations on the registers as occurring over possibly-overlapping intervals of time, they are treated as occurring instantaneously. The history of an execution of the system can thus be described simply as a sequence of operations. Concurrency in the system as a whole is modeled by the interleaving of operations from different processors in this sequence.

The actual order of the interleaving is the primary source of nondeterminism in the system. At any given time there may be up to n processors that are ready to execute another operation; how, then, does the system choose which of the processors will run next? We would like to make as few assumptions here as possible, so that our protocols will work under the widest possible set of circumstances. One way of doing this is to assign control over timing to an **adversary scheduler**, a function that chooses a processor to run at each step based on the previous history and current state of the system. The adversary scheduler is not bound by any fairness constraints; it may start and stop processors at will, doing whatever is necessary to prevent a protocol from executing correctly. In addition, no limits are placed on the scheduler's computational power or knowledge of the programming or internal states of the processors. However, its control is limited only to the timing of events in the system— it cannot, for example, cause a read operation to return the

sors ever again. However, there is no reason to believe that dead processors are the only source of difficulty in an asynchronous environment. For example, the adversary could choose to put some processor to sleep for a very long interval, waking it only when its view of the world was so outdated that its misguided actions would only hinder the completion of a protocol. As the hallway example in the introduction shows, stopping a processor and reawakening it much later can be even more devastating stopping a processor forever. Furthermore, distinguishing between slow processors and dead ones requires either an assumption that slow processors must take a step after some bounded interval, or that fast processors may execute a potentially unbounded number of operations waiting for the slow processors to revive. The first assumption imposes a weak form of synchrony on the system, violating the principle of avoiding helpful assumptions; the second makes it difficult to measure the efficiency of a protocol. For these reasons we avoid the issue completely by using the more general definition.

Other alternatives to the model involve changing the underlying communications medium from atomic registers, either by adopting stronger primitives that provide greater synchronization, or by moving to some sort of message-passing model. We avoid the first approach because, as always, we would like to work in as weak a model as possible. However, the question of how a different choice of primitives can affect the difficulty of solving wait-free consensus is an interesting one about which little is known, except for the deterministic case [LAA87, Her91].

Moving to a message-passing model presents new difficulties. In general, the defining property of a message-passing model is that the processors communicate by sending messages to each other directly, rather than operating on a common pool of registers or other primitives. Message-passing models come in bewildering variety; a general taxonomy can be found in [LL90]. Dolev et al. [DDS87] classify a large collection of message-passing models and show which are capable of solving consensus deterministically.

Among these many models, one has traditionally been associated with solving asynchronous consensus [BND89, BT83, CM89, FLP85]. In this model, the adversary is allowed to (i) stop up to t processors and (ii) delay messages arbitrarily. Unfortunately, a simple partition argument shows that in this model one cannot solve consensus even with a randomized algorithm if at least $n/2$ processors can fail [BT83]. Intuitively, the adversary can divide the processors into two groups of size $n/2$ and delay all messages

text, I will concentrate primarily on it. However, because the total-work measure has traditionally been used to analyze consensus protocols it will be considered as well.

An alternative to these measures that has seen some use in analyzing wait-free protocols is the **rounds** measure of asynchronous time [AFL83, ALS90, LF81, SSW91]. It is used for models that represent halting failures explicitly. When using this measure, up to $n-1$ processes may be designated as faulty at the discretion of the adversary; once a processor becomes faulty it is never allowed to execute another operation. A round is a minimal interval during which every non-faulty processor executes at least one operation. The measure is simply the number of these rounds. In effect, this measure counts the operations of the slowest non-faulty processor at any given point in the execution. If a slow processor executes only one operation in a given interval, only one round has elapsed, even though a faster processor might have carried out hundreds of operations during the same interval.

The rounds measure is reasonable if one defines the property of being wait-free as equivalent to being able to survive up to $n-1$ halting failures. However, as explained above, in the context of a totally asynchronous system this definition is unnecessarily restrictive. But once we adopt the more general definitions we quickly run into trouble. If some processor stops and then starts again much later during the execution of the protocol, the entire period that the processor is inactive counts as only one round. As a result the rounds measure implicitly resolves the problem of distinguishing slow processors from dead ones by guaranteeing that processors will either run at bounded relative speeds or not run at all. This is in conflict with the goal of using a model that is as general as possible, and for this reason the rounds measure will not be used here.[1]

[1] A notion of "rounds" does appear in Section 3.3; these rounds are part of the internal structure of the protocol described there and have no relation to the rounds measure.

many possible models; in the asynchronous shared-memory model it translates into requiring that the protocol be **wait-free**, as it requires that processors must finish in finite expected time regardless of the actions of the adversary scheduler.

If it happens that the processors already agree with each other, we want the consensus protocol to ratify that agreement rather than veto it; hence the validity condition. From a less practical perspective the validity condition is needed because its absence makes the problem uninteresting, since all of the processors could just decide 0 every time the protocol is run without any communication at all.

If we are allowed to make convenient assumptions about the system, consensus is not a difficult problem. For example, on a PRAM (perhaps the friendliest cousin of asynchronous shared-memory) consensus reduces to simply taking any function we like of the input values that satisfies the validity condition. In general, in any model where both the processors and the communications medium are reliable the problem can be solved simply by having the processors exchange information about their inputs until all of them know the entire set of inputs; at this point each can individually compute a function of the inputs as in the PRAM case to come up with the decision value for the protocol. It is only when we move to a model, like asynchronous shared-memory, that allows processors to fail that consensus becomes hard.

One difficulty is that the harsh assumptions of the asynchronous shared-memory model can amplify the correctness conditions in ways that may not be immediately obvious. For example, the validity conditions implies that the adversary can always force the processors to decide on a particular value by running only those processors that started with that value. Because these "live" processors are unable to see the differing input values of the "dead" processors, they will see a situation indistinguishable from one in which every processor started with the same value. In this latter case, the validity condition would force the processors to decide on that common value. So because of their limited knowledge, the live processors must decide on the only input value they can see, even though there may be other processors that disagree with it. This example shows that one must be very careful about what assumptions one makes in the model, as they can subtly affect what a protocol is allowed to do.

is the **bias**, ϵ, defined by $\epsilon = 1/2 - \delta$. In terms of the bias the agreement property can be restated as follows:

- **Bounded bias.** The probability that at least one processor decides on a given value is at most $1/2 + \epsilon$.

This property says in effect that the adversary can force some processor to see a particular outcome with only ϵ greater probability than if the processors were actually collectively flipping a fair coin.

In some circumstances we would like to guarantee that all of the processors *always* agree on the outcome of the coin, even though the adversary might have been able to control what that outcome is. A shared coin that guarantees agreement will be called **robust**. As will be seen in Chapter 4, robust shared coins can often be converted directly into consensus protocols by the addition of only a small amount of machinery. However Chapter 5 describes an intrinsically non-robust shared coin; in this situation more sophisticated techniques are needed to achieve consensus. One approach is described in the next section.

3.3 Consensus using shared coins

It is a well-established result that one can construct a consensus protocol from a shared coin with constant agreement parameter [ADS89, AH90a, SSW91]. This section gives as an example the first of these constructions [AH90a]. As we shall see, this construction gives a consensus protocol which requires an expected $O((T(n) + n)/\delta)$ operations per processor and $O((T'(n) + n^2)/\delta)$ total operations, where $T(n)$ and $T'(n)$ are the expected number of operations per processor and total operations for the shared coin protocol.

Pseudocode for each processor's behavior in the shared-coin-based consensus protocol is given in Figure 3.1. Each processor has a register of its own with two fields: *prefer* and *round*, initialized to $(\perp, 0)$. In addition there are assumed to be a (potentially unbounded) collection of shared coin primitives, one for each "round" of the protocol. Two special terms are used to simplify the description of the protocol. A processor is a **leader** if its *round* field is greater than or equal to every other process's *round* field. Two processors **agree** if both their *prefer* fields are equal, and neither field is $\perp$.

details of the protocol. The interested reader is referred to [AH90a] for a more thorough description of the construction including a full proof of correctness. Alternative constructions with similar performance may be found in [ADS89] and [SSW91].

For our purposes it will suffice to summarize the relevant results from [AH90a]:

Theorem 3.1 **([AH90a])** *The protocol of Figure 3.1 implements a consensus protocol that requires an expected $O(1/\delta)$ rounds, where δ is the agreement parameter of the shared coin.*

From which it follows that:

Corollary 3.2 *The protocol of Figure 3.1 implements a consensus protocol that requires an expected $O((T(n) + n)/\delta)$ operations per processor and $O((T'(n) + n^2)/\delta)$ operations in total; where δ is the agreement parameter, $T(n)$ the expected number of operations per processor, and $T'(n)$ the expected number of operations in total for the shared coin.*

Proof: From the theorem, we expect at most $O(1/\delta)$ rounds.

In each round, each processor executes at most $2n$ read operations, one instance of the shared coin, and two write operations, for a total of $2n + 2 + T(n)$ operations.

Similarly, in each round the processors collectively execute at most $2n^2$ read operations, $2n$ write operations, and one instance of the shared coin, for a total of $2n^2 + 2n + T'(n)$ operations. ∎

decide 0. So to use a simple random-walk-based shared coin in a consensus protocol one would need to run it repeatedly as described in Section 3.3.

The protocols described in this chapter avoid the need for such methods by extending the random walk to incorporate the function of detecting agreement. As a result we obtain a *robust* shared coin, described in Section 4.2, which guarantees that all processors agree on its outcome. Because the coin guarantees agreement, it can be modified in to obtain a consensus protocol simply by attaching a preamble to ensure validity, as described in Section 4.4. The resulting consensus protocol (and its variants, obtained by replacing the counter implementation [BR90, DHPW92]) are particularly simple, as they are the only known wait-free consensus protocols that do not require the repeated execution of a non-robust shared coin protocol and the multi-round superstructure that comes with it.

The simplicity of the protocol also allows some optimizations that are more difficult when using a non-robust coin. The consensus protocol is designed to require fewer total operations if fewer processors actually participate in it, a feature which becomes important when, for example, the protocol is used as a primitive for building shared data structures which only a few processors might attempt to access simultaneously.

The chapter is organized as follows. Section 4.1 describes some properties of random walks that will be used later in the chapter. Section 4.2 describes the robust shared coin protocol and proves its correctness. The description of the robust shared coin protocol assumes the presence of an **atomic counter**, providing increment, decrement, and read operations that appear to occur sequentially; Section 4.3 shows how such a counter may be built from single-writer atomic registers at the cost of $O(n^2)$ register operations per counter operation. Finally, Section 4.4 describes the consensus protocol obtained by modifying the robust shared coin.

4.1 Random walks

Let us begin by stating a few basic lemmas about the behavior of random walks.

Lemma 4.1 *Consider a symmetric random walk with step size 1 running between absorbing barriers at a and b and starting at x, where $a < x < b$. Then:*

points as dividing the range of the random walk into intervals; between each pair of points where the adversary forces the particle to move deterministically is a region where the particle moves randomly. The points at the edge of these random regions act like barriers in a random walk. A point on the side away from c pushes the particle into a new region and so acts like an absorbing barrier, while a point on the side toward c pushes the particle back into the old region and so acts like a reflecting barrier. Thus the region containing c acts like a random walk with two absorbing barriers, and the remaining regions act like random walks with one absorbing barrier (on the side away from c) and one reflecting barrier (on the side toward c).

Because each barrier can only be crossed away from c, once the particle leaves a region it can never return. Now, suppose the particle starts in a region with width w_1. After at most w_1^2 steps on average (by Lemmas 4.1 or 4.2) it will pass into a new region of width w_2; after an additional w_2^2 steps it will pass into a new region of width w_3, and so on until either a or b is reached. Since these regions all fit between a and b, $\sum w_i \leq b - a$, and thus (since each $w_i > 0$) $\sum w_i^2 \leq (b-a)^2$. ∎

Though the bound in Lemma 4.3 is proved for the case of a very powerful adversary that is always allowed to choose between a random move and a deterministic move at each step, the bound applies equally well to a weaker adversary whose choices are more constrained, as the stronger adversary could always choose to operate within the weaker adversary's constraints. This technique, of proving bounds for a strong adversary that carry over to a weaker one, has great simplifying power. It will be used extensively in the analysis of the shared coin and consensus protocols.

4.2 The robust shared coin protocol

Figure 4.1 shows pseudocode for each processor's behavior in the robust shared coin protocol. The coin is constructed using an **atomic counter**, which supports atomic increment, decrement, and read operations. In this section, these operations are assumed to take unit time. The counter is initialized to 0. The processor's local coin is represented by the procedure *local_flip*, which returns the values -1 and 1 with equal probability.

A processor's behavior in the protocol is represented in pictorial form in Figure 4.2. While a processor reads values in the central range from $-K$

to K (where K is a parameter of the protocol) it flips a local fair coin to decide whether to increment or decrement the counter. This part of the protocol is essentially the same as the random-walk-based shared coin of Aspnes and Herlihy [AH90a]. What is new is the addition of a "slope" at either side of the random walk. On these slopes, a processor does not move the counter randomly but instead always moves it away from the center. When a processor reads a counter value in one of the "buckets" beyond the slopes, it decides either 0 or 1 depending on the sign of the counter.

If the slopes are wide enough, once any processor has seen a value that causes it to decide, all other processors will see values that cause them to push the counter toward the same decision. This mechanism eliminates the possibility that delayed writes might move the counter out of the decision range and allow the random walk (with small but non-negligible probability) to wander over to the other side. More formally, we can show:

Lemma 4.4 *If any processor reads a counter value $v \geq (K+n)$, then all subsequent reads will return values greater than or equal to $K+1$; in the symmetric case where $v \leq -(K+n)$, all subsequent values read will be less than or equal to $-(K+1)$.*

Proof: Suppose that a processor has read $v \geq (K+n)$; then it immediately terminates leaving $n-1$ running processors. Thus the number d of processors that will execute a decrement before their next read is at most $n-1$. Let $l = c - d$ where c is the value stored in the counter. Since $c \geq (K+n)$, it must be the case that $l \geq K+1$. Now consider the effect of the actions the scheduler can take. If it allows a decrement to proceed, c and d both drop by 1 and l remains constant. If it allows an increment to occur, c increases and l increases with it. If it allows a read, the value read is $c \geq l \geq K+1$, and thus d is unaffected. In each case l remains at least $K+1$, and the claim follows since $c \geq l$. The proof of the symmetric case is similar. ∎

The consistency property follows immediately from Lemma 4.4. A similar argument shows that the counter will not overflow:

Lemma 4.5 *The counter value never leaves the range $[K-3n, K+3n]$ in any execution of the shared coin protocol.*

Proof: Suppose that the counter reaches $K+2n$ at some point. Then each processor will execute at most one increment or decrement operation before

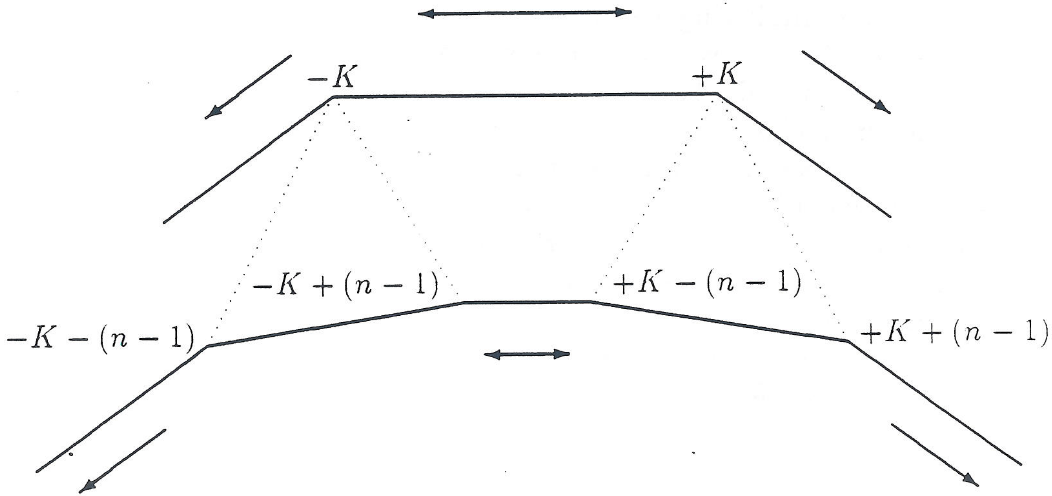


Figure 4.3: The protocol as a controlled random walk.

Lemma 4.7 *The robust shared coin protocol executes an expected $O((K+n)^2)$ total counter operations when $K \geq n$.*

Proof: If we consider the true position t, Lemma 4.6 implies that the scheduler can only force t up if $t \geq K-(n-1) \geq 1$ and down if $t \leq -K+(n-1) \leq -1$. Furthermore if $|t|$ ever exceeds $K+n+(n-1)$, each processor will decide after its next read. Thus the movement of the true position is a controlled random walk in the sense of Lemma 4.3 with center 0 and barriers at $\pm(K+2n-1)$. The expected number of steps until a barrier is reached is at most $4(K+2n-1)^2$ steps, which will be followed by at most $2n$ operations as the processors each decide. Since each step takes a constant number of counter operations the expected number of operations required is $O((K+n)^2)$. ∎

The time bound of Lemma 4.7 shows that every processor terminates in finite expected time when $K \geq n$. The bounded bias property is a consequence of the following lemma:

Lemma 4.8 *Against any scheduler, the probability that the processors in the robust shared coin protocol will decide 1 is between $\frac{K-(n-1)}{2K}$ and $\frac{K+(n-1)}{2K}$.*

4.3 Implementing a bounded counter with atomic registers

The robust shared coin protocol assumes the presence of a shared counter supporting atomic increment, decrement, and read operations, with the restriction that no operation will be applied that will move the counter out of some fixed range $[-r, r]$. In practice such a counter is not likely to be available as a hardware primitive. Fortunately it is not difficult to implement a shared counter using atomic registers. However, some care must be taken to guarantee that the counter uses only a bounded amount of space.

Both Aspnes and Herlihy [AH90a] and Attiya, Dolev, and Shavit [ADS89] describe shared counter implementations. The two counter implementations both assign a register to hold the net increment due to each processor, so that the counter's value is simply the sum of the values in these registers. Both algorithms use simple atomic snapshot protocols to allow the entire set of registers to be read in a single atomic action.

Alas, neither implementation does quite what we would like. Even though the value stored in the counter will never exceed the range $[-r, r]$, the net increment due to an individual processor is potentially unbounded. The Aspnes-Herlihy protocol ignores this difficulty by assuming the presence of unbounded registers (which it also uses to implement the atomic scan.) The Attiya-Dolev-Shavit protocol uses only bounded registers, but enforces the bounds by prematurely terminating the shared coin protocol if any processor's register wanders out of a limited range. This premature termination occurs infrequently, and is acceptable in a shared coin that does not need to guarantee consistency. But it is not acceptable for a robust coin, as it may allow the scheduler to force some processor to choose one value (through premature termination) after another has already chosen a different value (through the normal workings of the shared coin protocol.)

A simple alternative to premature termination that still allows the size of the registers to be bounded is to store the remainder of each processor's contribution relative to some convenient modulus m greater than the total range $2r + 1$. The counter value can then be reconstructed as the unique v in the range $[-r, r]$ that is congruent to the sum of the registers, modulo m. Pseudocode for the three counter operations using this technique is shown in Figure 4.4; it assumes the presence of an array of registers which can be

4.4 The randomized consensus protocol

Figure 4.5 shows pseudocode for each processor's behavior in the randomized consensus protocol. The protocol uses three shared counters. The first two maintain a total of the number of participating processors that started with each of the inputs 0 and 1. The last is used as the counter for a modified version of the robust shared coin protocol. All of the counters have an initial value of 0.

The protocol is optimized for the case where few processors participate. We will define a processor to be **active** if it takes at least one step before some processor decides on a value, and denote by p the total number of active processors in a given execution. The protocol uses the counters a_0 and a_1 to keep track of the number of active processors by having each processor increment one or the other of these counters as it starts the protocol.

The protocol depends on being able to take an atomic snapshot of the counters. Since the first two counters are never decremented, such a snapshot can be obtained as described in Figure 4.6. Though the operation defined there is not wait-free, because it will not finish if a_0 or a_1 changes during some pass through the loop, this event can occur at most p times during any execution of the consensus protocol. So in fact the time to carry out the atomic snapshot will be bounded in the context in which it is used.

If the counters are not primitives but are instead constructed as described in Section 4.3 using an atomic scan operation, the overhead of Figure 4.6 can be avoided completely by simply reading all three counters in a single atomic scan of the arrays that implement them.

Several features of the protocol are worth noting. First of all, the same "slopes" that ensured consistency for the robust shared coin ensure consistency for the consensus protocol, for the same reasons. Second, the counters a_0 and a_1 allow the protocol to guarantee validity, as the random walk is only invoked if both have non-zero values. These counters are also used to minimize the range of the random walk, by taking advantage of the fact stated in the following lemma, a modification of Lemma 4.6:

Lemma 4.10 *Let a_0, a_1, c be the values read from the counters by some processor and t the true position of the random walk in the state preceding the read. Then $|c - t| \leq a_0 + a_1 - 1$.*

Proof: There are at most $a_0 + a_1 - 1$ processors with pending increments or

decrements. ∎

To prove that the consensus protocol is correct, we must establish that it is consistent, that it terminates, and that it is valid. The proof of consistency is a straightforward modification of the proof of Lemma 4.4:

Lemma 4.11 *If any processor reads a counter value $v \geq 2n$, then all subsequent reads will return values $\geq n+1$; in the symmetric case where $v \leq -2n$, all subsequent reads will return values $\leq -(n+1)$.*

Proof: Apply the proof of Lemma 4.4 with $K = n$. ∎

Similarly, the proof that the counter c does not overflow is a straightforward modification of Lemma 4.5:

Lemma 4.12 *The value of c never leaves the range $[-4n, 4n]$ in any execution of the consensus protocol.*

Proof: Apply the proof of Lemma 4.5 with $K = n$. ∎

Termination is trickier to demonstrate. As in the case of the shared coin, the key to proving the consensus protocol's termination is the fact that the scheduler's only alternative to moving the true position randomly is to move it away from the origin. In the shared coin protocol, this condition depends on fixing the parameter $K \geq n$. In the consensus protocol the situation is more complicated, as the protocol uses its knowledge of the number of currently active processors to set the inner boundaries of the slope close to the origin while still preventing the scheduler from being able to force the true position to move toward the origin.

Lemma 4.13 *Let n be the total number of processors and p be the number of processors that take at least one step before some processor decides on a value. Then the worst-case expected running time of the consensus protocol is $O(p^2 + n)$ total counter operations.*

Proof: We will show that the consensus protocol terminates in $O(p^2 + n)$ time by reducing it to a controlled random walk of the true position t. Divide the execution of the protocol into two phases. In the first phase, at most one of a_0, a_1 is nonzero; if the execution does not leave the first phase before

Proof: Lemmas 4.11, 4.13, and 4.14. ∎

It is worth looking at the behavior of the shared coin implicitly embedded in the consensus protocol of Figure 4.5. Because the function of detecting agreement is implemented in the shared coin itself, limiting scheduler control over the outcome of the shared coin is no longer necessary to achieve consensus. Thus the parameter K of the shared coin protocol can be set to minimize the time taken in the random walk without regard to its effect on the agreement parameter δ. In the protocol of Figure 4.5 the shared coin has an effective agreement parameter of $\frac{1}{2p}$, as low as is possible without setting $K < p$.

At the same time, the simplicity of the protocol allows the number and size of the shared counters to be very small. Unfortunately, when the available primitives are limited to atomic registers this small size is lost in the $\Theta(n^2)$ space overhead of the atomic scan operation. It is not immediately clear that this overhead is a necessary feature of an atomic counter implementation; much work remains to be done in this area.

multiple of n^2 votes are cast, the total variance will be $\Omega(n^2)$. Because the total vote is approximately normally distributed, the protocol can guarantee that with constant probability the total vote is more than n away from the origin, rendering the scheduler's adjustment ineffective.

Alas, the very anonymity of the processors that is the strength of the voting technique is also its greatest weakness. To overcome the scheduler's power to withhold votes, it is necessary that a total of $\Omega(n^2)$ votes are cast—but the scheduler might also choose to stop all but one of the processors, leaving that lone processor to generate all $\Omega(n^2)$ votes by itself. It follows that, for all of the polynomial-time wait-free consensus protocols based on unweighted voting, the worst-case expected bound on the work done by a single processor is asymptotically no better than the bound on the total work done by all of the processors together.

In this chapter we show how to avoid this problem by modifying a protocol of Bracha and Rachman [BR91] to allow the processor to cast votes of increasing weight. Thus a fast processor or a processor running in isolation can quickly generate votes of sufficient total variance to finish the protocol. at the cost of giving the scheduler greater control by allowing it both to withhold votes with larger impact and to choose among up to n different weights (one for each processor) when determining the weight of the next vote.

There are two main difficulties that this approach entails. The first is that careful adjustment of the weight function and other parameters of the protocol is necessary to make sure that it performs correctly. More importantly, allowing the weight of the i-th vote to depend on the particular processor the scheduler chooses to run, which may in turn depend on the outcomes of previous votes, means that we cannot treat the sequence of votes as a sequence of independent random variables.

However, the *sign* of each vote is determined by a fair coin flip that the scheduler cannot predict in advance, and so despite all the scheduler's powers, the expected value of each vote before it is cast is always 0. This is the primary requirement of a **martingale process** [Bil86, Fel71, Kop84]. Under the right conditions, martingales have many similarities to sequences of sums of independent random variables. In particular, martingale analogues of the Central Limit Theorem and Chernoff bounds will be used in the proof of correctness.

The rest of the chapter is organized as follows. Section 5.2 defines the shared coin protocol and gives an overview of its operation. Section 5.3

each flip is $w(t)$ or $-w(t)$ respectively, where t is the number of coins flipped by the processor up to and including its current flip. Each weighted flip represents a vote for either the output value 1 (if positive) or 0 (if negative). After each flip, the processor updates its register to hold the sum of the weighted flips it has performed, and the sum of the squares of their values. After every c flips, the processor reads the registers of all the other processors, and computes the sum of all the weighted flips (the total vote) and the sum of the squares of their values (the total variance). If the total variance is greater than the quorum K, it stops, and outputs 1 if the total vote is positive, and 0 if it is negative (it treats a total vote of zero as a failure to avoid introducing asymmetry between the two outcomes). Alternatively, if the total variance has not yet reached the quorum K, it continues to flip its local coin.

As in the previous chapter, the function *local_flip* returns the values 1 and -1 randomly with equal probability. The values K and c are parameters of the protocol which will be set depending on the number of processors n to give the desired bounds on the agreement parameter and running time. The weight function $w(t)$ is used to make later local coin flips have more effect than earlier ones, so that a processor running in isolation will be able to achieve the quorum K quickly. The weight function will be assumed to be of the form $w(t) = t^a$ where a is a nonnegative parameter depending on n; though other weight functions are possible, this choice simplifies the analysis.

We will demonstrate that for suitable choice of K, c and a all processors return 1 with constant probability; the case of all processors returning 0 will follow by symmetry. The structure of the argument follows the proof of correctness of the less sophisticated protocol of Bracha and Rachman [BR91], which corresponds to Figure 5.1 when $w(t)$ is the constant 1, $K = \Theta(n^2)$, and $c = \Theta(n/\log n)$. Votes cast before the quorum K is reached will form a pool of **common votes** that all processors see.[1] We will show that with constant probability (i) the total of the common votes is far from the origin and (ii) the sum of the **extra votes** cast between the time the quorum is reached and the time some processor does its final read in line 13 is small, so that the total vote read by each processor will have the same sign as the total common vote.

[1]The definitions of the common and extra votes we will use differ slightly from those used in [BR91]; the formal definitions appear in Section 5.4.

additional knowledge does not affect her ability to predict the future. To do so, the definition of a martingale must be extended to allow additional information to be represented explicitly.

The tool used to represent the information known at any point in time will be a concept from measure theory, a **σ-algebra**[2] The description given here is informal; more complete definitions can be found in [Fel71, Sections IV.3, IV.4, and V.11] or [Bil86].

5.3.1 Knowledge, σ-algebras, and measurability

Recall that any probabilistic statement is always made in the context of some (possibly implicit) **sample space**. The elements of the sample space (called **sample points**) represent all possible results of some set of experiments, such as flipping a sequence of coins or choosing a point at random from the unit interval. Intuitively, all randomness is reduced to selecting a single point from the sample space. An **event**, such as a particular coin-flip coming up heads or a random variable taking on the value 0, is simply a subset of the sample space that "occurs" if one of the sample points it contains is selected.

If we are omniscient, we can see which sample point is chosen and thus can tell for each event whether it occurs or not. However, if we have only partial information, we will not be able to determine whether some events occurred or not. We can represent the extent of our knowledge by making a list of all events we do know about. This list will have to satisfy certain closure properties; for example, if we know whether or not A occurred, and whether or not B occurred, then we should know whether or not the event "A or B" occurred.

We will require that the set of known events be a **σ-algebra**. A σ-algebra $\mathcal{F}$ is a family of subsets of a sample space Ω that (i) contains the empty set; (ii) is closed under complement: if $\mathcal{F}$ contains A, it contains $\Omega \setminus A$ (the **complement** of A); and (iii) is closed under countable union: if $\mathcal{F}$ contains all of $A_1, A_2, \ldots$, it contains $\bigcup_{i=1}^{\infty} A_i$.[3] An event A is said to be **$\mathcal{F}$-measurable** if it is contained in $\mathcal{F}$. In our context, the term "measurable," which comes from the original measure-theoretic use of σ-algebras to represent families of sets on which a probability distribution is well-defined, simply means "known."

---

[2]Sometimes called a **σ-field**.

[3]Additional properties, such as being closed under finite union or intersection, follow immediately from this definition.

$\mathcal{F}_i$ is a σ-algebra representing the knowledge of the underlying probability distribution available at time i. Martingales are required to satisfy three axioms, for all i:

1. $\mathcal{F}_i \subseteq \mathcal{F}_{i+1}$. (The past is never forgotten.)
2. S_i is $\mathcal{F}_i$-measurable. (The present is always known.)
3. $\mathrm{E}[S_{i+1} \mid \mathcal{F}_i] = S_i$. (The future cannot be foreseen.)

Often $\mathcal{F}_i$ will simply be the σ-algebra $\langle S_1, \ldots S_i \rangle$ generated by the variables S_1 through S_i; in this case axioms 1 and 2 will hold automatically.

To avoid special cases let $\mathcal{F}_0$ denote the trivial σ-algebra consisting of the empty set and the entire probability space. The **difference sequence** of a martingale is the sequence $X_1, X_2, \ldots X_n$ where $X_1 = S_1$ and $X_i = S_i - S_{i-1}$ for $i > 1$. A **zero-mean martingale** is a martingale for which $\mathrm{E}[S_i] = 0$.

5.3.3 Gambling systems

A remarkably useful theorem, which has its origins in the study of gambling systems, is due to Halmos [Hal39]. We restate his theorem below in modern notation:

Theorem 5.1 *Let* $\{S_i, \mathcal{F}_i\}, 1 \leq i \leq n$ *be a martingale with difference sequence* $\{X_i\}$. *Let* $\{\zeta_i\}, 1 \leq i \leq n$ *be random variables taking on the values* 0 *and* 1 *such that each* ζ_i *is* $\mathcal{F}_{i-1}$*-measurable. Then the sequence of random variables* $S'_i = \sum_{j=1}^{i} \zeta_j X_j$ *is a martingale relative to* $\mathcal{F}_i$.

Proof: The first two properties are easily verified. Because ζ_i is $\mathcal{F}_{i-1}$-measurable, $\mathrm{E}[\zeta_i X_i \mid \mathcal{F}_{i-1}] = \zeta_i \mathrm{E}[X_i \mid \mathcal{F}_{i-1}] = 0$, and the third property also follows. ∎

5.3.4 Limit theorems

Many results that hold for sums of independent random variables carry over in modified form to martingales. For example, the following theorem of Hall and Heyde [HH80, Theorem 3.9] is a martingale version of the classical Central Limit Theorem:

If $n = 1$ we are done, since $w_1^2 \le W$. If n is greater than 1, for each $i \le n-1$ let $S'_i = S_{i+1} - X_1$ and $\mathcal{F}'_i = \mathcal{F}_{i+1}$. Then $\{S'_i, \mathcal{F}'_i\}, 1 \le i \le n-1$ satisfies the conditions of the lemma with $\mathcal{F}'_0 = \mathcal{F}_1$, $w'_i = w_{i+1}$ and $W' = W - w_1^2$, so by the induction hypothesis $\mathrm{E}\left[e^{\alpha S'_{n-1}} \mid \mathcal{F}'_0\right] \le e^{\alpha^2(W-w_1^2)/2}$. But then, using the fact that $\mathrm{E}[X \mid \mathcal{F}] = \mathrm{E}[\mathrm{E}[X \mid \mathcal{F}'] \mid \mathcal{F}]$ when $\mathcal{F} \subseteq \mathcal{F}'$, we can compute:

$$\begin{aligned}
\mathrm{E}\left[e^{\alpha S_n} \mid \mathcal{F}_0\right] &= \mathrm{E}\left[\mathrm{E}\left[e^{\alpha X_1} e^{\alpha(S_n - X_1)} \mid \mathcal{F}_1\right] \mid \mathcal{F}_0\right] \\
&= \mathrm{E}\left[e^{\alpha X_1} \mathrm{E}\left[e^{\alpha S'_{n-1}} \mid \mathcal{F}'_0\right] \mid \mathcal{F}_0\right] \\
&\le \mathrm{E}\left[e^{\alpha X_1} e^{\alpha^2(W-w_1^2)/2} \mid \mathcal{F}_0\right] \\
&= e^{\alpha^2(W-w_1^2)/2} \mathrm{E}\left[e^{\alpha X_1} \mid \mathcal{F}_0\right] \\
&\le e^{\alpha^2(W-w_1^2)/2} e^{\alpha^2 w_1^2/2} \\
&= e^{\alpha^2 W/2}.
\end{aligned}$$

∎

Theorem 5.4 *Let $\{S_i, \mathcal{F}_i\}, 1 \le i \le n$ be a zero-mean martingale with difference sequence $\{X_i\}$. If there exists a sequence of random variables $w_1, w_2, \ldots w_n$, and a constant W, such that*

1. *Each w_i is $\mathcal{F}_{i-1}$-measurable.*
2. *For all i, $|X_i| \le w_i$ with probability 1, and*
3. *$\sum_{i=1}^n w_i^2 \le W$ with probability 1,*

then for any $\lambda > 0$,

$$\Pr[S_n \ge \lambda] \le e^{-\lambda^2/2W}. \tag{5.3}$$

Proof: By Lemma 5.3, for any $\alpha > 0$, $\mathrm{E}\left[e^{\alpha S_n}\right] \le e^{\alpha^2 W/2}$. Thus by Markov's inequality

$$\Pr[S_n \ge \lambda] = \Pr\left[e^{\alpha S_n} \ge e^{\alpha\lambda}\right] \le e^{\alpha^2 W/2} e^{-\alpha\lambda}.$$

Setting $\alpha = \lambda/W$ gives (5.3). ∎

Symmetry immediately gives us:

vote X_i is cast, and thus both κ_i and $\zeta_{P,i}$ are $\mathcal{F}_{i-1}$-measurable. Consequently the sequences $\left\{\sum_{j=1}^{i} \kappa_j X_j\right\}$ and $\left\{\sum_{j=1}^{i} \zeta_{P,j} X_j\right\}$ are martingales relative to $\{\mathcal{F}_i\}$ by Theorem 5.1. Votes for which $\zeta_{P,i} = 1$ but $\kappa_i = 0$ will be referred to as the **extra votes** for processor P. (Observe that $\zeta_{P,i} \geq \kappa_i$ since P could not have started its final read until the total variance was at least K.) The sequence $\left\{\sum_{j=1}^{i} (\zeta_{P,i} - \kappa_i) X_i\right\}$ of the partial sums of these extra votes is a difference of martingales and is thus also a martingale relative to $\{\mathcal{F}_i\}$.

The structure of the proof of correctness is as follows. First, we show that the distribution of the total common vote, $\sum \kappa_i X_i$, is close to a normal distribution with mean 0 and variance K for suitable choices of a and K; in particular, for n sufficiently large, the probability that $\sum \kappa_i X_i > x\sqrt{K}$ will be at least a constant for any fixed x. Next, we complete the proof by showing that if the total common vote is far from the origin the chances that any processor will read a total vote whose sign differs from the common vote is small. This fact is itself shown in two steps. First, it is shown that, for suitable choice of c, the total of the extra votes for a processor P, $\sum(\zeta_{P,i} - \kappa_i) X_i$, will be small with high probability. Second, a bound Δ is derived on the difference between $\sum \zeta_{P,i} X_i$ and the total vote actually read by P.

It will be necessary to select values for a, K, and c that give the correct bounds on the probabilities. However, we will be in a better position to justify our choice for these parameters after we have developed more of the analysis, so the choice of parameters will be deferred until Section 5.4.5.

5.4.1 Phases of the protocol

We begin by defining the phases of the protocol more carefully. Let t_i be the value of the i-th processor's internal variable t at any given step of the protocol. Let U_i be the random variable representing the maximum value of t_i during the entire execution of the protocol. Let T_i be the random variable representing the maximum value of t_i during the part of the execution of the protocol where $\kappa_i = 1$.

In the proof of correctness we will encounter many quantities of the form $\sum_{i=1}^{n} \xi(T_i)$ or $\sum_{i=1}^{n} \xi(U_i)$ for various functions ξ. We will want to get bounds on these quantities without having to look too closely at the particular values of each T_i or U_i. This section proves several very general inequalities about

Lemma 5.8 *Let $\psi(x) = x^A/A$ and let χ be any strictly increasing function such that $\chi\psi^{-1}$ is convex. Then for any non-negative $\{x_i\}$, if $\sum_{i=1}^n x_i^A/A \leq K$, then $\sum_{i=1}^n \chi(x_i) \leq (n-1)\chi(0) + \chi(n^{1/A}T_K)$.*

Proof: Let $Y = \sum \psi(x_i)$. Now $\chi(x_i) = \chi\psi^{-1}\psi(x_i)$ or

$$\chi\psi^{-1}\left(\left(1 - \frac{\psi(x_i)}{Y}\right)0 + \frac{\psi(x_i)}{Y}Y\right)$$

which is at most

$$\left(1 - \frac{\psi(x_i)}{Y}\right)\chi\psi^{-1}(0) + \frac{\psi(x_i)}{Y}\chi\psi^{-1}(Y)$$

given the convexity of $\chi\psi^{-1}$. Hence

$$\begin{aligned}
\sum_{i=1}^n \chi(x_i) &\leq n\chi\psi^{-1}(0) - \left(\sum_{i=1}^n \frac{\psi(x_i)}{Y}\right)\chi\psi^{-1}(0) + \left(\sum_{i=1}^n \frac{\psi(x_i)}{Y}\right)\chi\psi^{-1}(Y) \\
&= (n-1)\chi\psi^{-1}(0) + \chi\psi^{-1}\left(\sum_{i=1}^n \psi(x_i)\right) \\
&\leq (n-1)\chi\psi^{-1}(0) + \chi\psi^{-1}(K)
\end{aligned}$$

which is just $(n-1)\chi(0) + \chi\left(n^{1/A}T_K\right)$. ∎

The quantity $n^{1/A}T_K$ is the maximum value that any x_i can take on without violating the constraint on $\sum x_i$. So what Lemma 5.8 says is that if $\chi\psi^{-1}$ is convex, $\sum \chi(x_i)$ is maximized by maximizing one of the x_i while setting the rest to zero.

For the variables U_i we can show:

Lemma 5.9 *Let $\psi(x) = x^A/A$ and let χ be any strictly increasing function such that $\chi(\psi^{-1}(x)+c+1)$ is concave in x. Then for any non-negative $\{x_i\}$, if $\sum_{i=1}^n \psi(x_i) \leq K$, then*

$$\sum_{i=1}^n \chi(U_i) \leq n\chi(T_K + c + 1) \tag{5.7}$$

where C_1 is an absolute constant.

Proof: The proof uses Theorem 5.2, which requires that the martingale be normalized so that the total conditional variance V_N^2 is close to 1. So let $Y_i = \frac{\kappa_i X_i}{\sqrt{K}}$ and consider the martingale $\left\{\sum_{j=1}^{i} Y_j, \mathcal{F}_i\right\}$. To apply the theorem we need to compute a bound on the value L_N. We will fix $\delta = 1$.

We begin by getting a bound on the first term $\sum \mathrm{E}\left[|Y_i|^{2+2\delta}\right]$. We have

$$\sum_{i=1}^{N} \mathrm{E}\left[|Y_i|^4\right] = \mathrm{E}\left[\sum_{i=1}^{N} |Y_i|^4\right] = \frac{1}{K^2}\mathrm{E}\left[\sum_{i=1}^{N} |\kappa_i X_i|^4\right] = \frac{1}{K^2}\mathrm{E}\left[\sum_{i=1}^{n}\sum_{j=1}^{T_i} j^{4a}\right] \quad (5.11)$$

Now,

$$\sum_{j=1}^{T_i} j^{4a} \leq \int_0^{T_i} j^{4a}\, dj + T_i^{4a} = \frac{T_i^{4a+1}}{4a+1} + T_i^{4a}.$$

Define $\psi(x) = x^A/A, \chi(x) = x^{4a} + \frac{x^{4a+1}}{4a+1}$, taking $0^0 = 1$. Then $\chi\psi^{-1}(y) = (Ay)^{4a/A} + \frac{(Ay)^{(4a+1)/A}}{4a+1}$ is convex, and hence $\sum_{i=1}^{n}\left(T_i^{4a} + \frac{T_i^{4a+1}}{4a+1}\right)$ is at most $(n^{1/A}T_K)^{4a} + \frac{(n^{1/A}T_K)^{4a+1}}{4a+1} + (n-1)\chi(0)$ using Lemma 5.8. If a is positive then $\chi(0)$ is zero; however if a is zero $\chi(0)$ will be 1. In either case $(n-1)\chi(0) \leq n-1$. Plugging everything value back into (5.11) gives

$$\sum_{i=1}^{N} \mathrm{E}\left[|Y_i|^4\right] \leq \frac{(n^{1/A}T_K)^{4a}}{K^2} + \frac{(n^{1/A}T_K)^{4a+1}}{K^2(4a+1)} + \frac{n-1}{K^2}. \quad (5.12)$$

For the second term $\mathrm{E}\left[|V_N^2 - 1|^{1+\delta}\right]$, observe that

$$V_N^2 = \sum_{i=1}^{N} \mathrm{E}\left[Y_i^2 \mid \mathcal{F}_{i-1}\right] = \frac{1}{K}\sum_{i=1}^{N} \mathrm{E}\left[(\kappa_i X_i)^2 \mid \mathcal{F}_{i-1}\right],$$

which is just $1/K$ times the sum of the squares of the weights $|X_i|$ of the common votes. But the total variance of the common votes can differ from K by at most the variance of the first vote X_i for which $\kappa_i = 0$. Since the processor that casts this vote can have cast at most $n^{1/A}T_K$ votes beforehand, the variance of this vote is at most $\left(n^{1/A}T_K + 1\right)^{2a}$, giving the bound

$$|V_N^2 - 1|^{1+\delta} \leq \frac{1}{K}\left(n^{1/A}T_K + 1\right)^{2a}. \quad (5.13)$$

could not have started its final read until the total variance exceeded K. As discussed above, both $\zeta_{P,i}$ and κ_i are $\mathcal{F}_{i-1}$-measurable. Thus $\xi_i = \zeta_{P,i} - \kappa_i$ is a $0-1$ random variable that is $\mathcal{F}_{i-1}$-measurable, and $\left\{S_{P,i} = \sum_{j=1}^{i} \xi_j X_j, \mathcal{F}_i\right\}$ is a martingale by Theorem 5.1.

Define $\Delta = n(gT_K)^a$. The following lemma shows a bound on the tails of $\sum \xi_i X_i$.

Lemma 5.12 *For any $x > 0$, if*

$$g^a \leq d\sqrt{\frac{T_K}{nA}} \tag{5.14}$$

holds for some positive $d < x$, and

$$g^A \leq 1 + \frac{(x-d)^2}{2\log(n/p)} \tag{5.15}$$

holds for some positive $p < n$, then for each processor P,

$$\Pr\left[\sum(\zeta_{P,i} - \kappa_i)X_i \leq \Delta - x\sqrt{K}\right] \leq p/n. \tag{5.16}$$

Proof: The proof uses Corollary 5.5, so we proceed by showing that its premises (stated in Theorem 5.4) are satisfied.

By Corollary 5.10, X_i and thus $\xi_i X_i$ is zero for $i > n(T_K + c + 1)$. So $\sum \xi_i X_i = S_{P,M}$ where $M = n(T_K + c + 1)$.

Set $w_i = |\xi_i X_i|$. Then the first premise of Corollary 5.5 follows from the fact that for each i, ξ_i and $|X_i|$ are both $\mathcal{F}_i$-measurable. The second premise is immediate. For the third premise, notice that

$$\sum(|\xi_i X_i|)^2 = \sum \xi_i X_i^2 = \sum \zeta_{P,i} X_i^2 - \sum \kappa_i X_i^2 \leq \sum X_i^2 - \sum \kappa_i X_i^2.$$

The first term is

$$\sum X_i^2 = \sum_{i=1}^{n} \sum_{j=1}^{U_i} j^{2a}.$$

The second term is

$$\sum \kappa_i X_i^2 \geq K - t^{2a}$$

But if (5.15) holds then

$$g^A - 1 \leq \frac{(x-d)^2}{2\log(n/p)}$$

and, since $\log(n/p) > 0$ and $g > 1$,

$$-\frac{(x-d)^2}{2(g^A-1)} \leq -\log(n/p) = \log(p/n),$$

From which it follows that

$$e^{-(x-d)^2/2(g^A-1)} \leq e^{\log(p/n)} = p/n.$$

∎

5.4.4 Written votes vs. decided votes

In this section we show that the difference between $\sum \zeta_{P,i} X_i$ and the total vote actually read by P is bounded by $\Delta = n(gT_K)^a$.

Lemma 5.13 *Let R_P be the sum of the votes read during P's final read. Then*

$$\left|\sum \zeta_{P,i} X_i - R_P\right| \leq n(T_k + c + 1)^a \leq n(gT_K)^a = \Delta \tag{5.20}$$

Proof: Suppose $\zeta_{P,i} = 1$, and suppose X_i is decided by processor P_j. If the vote X_i is not included in the value read by P, it must have been decided before P's read of P_j's register but written afterwards. Because each vote is written out before the next vote is decided there can be at most one vote from P_j which is included in $\sum \zeta_{P,i} X_i$ but is not actually read by P. This vote has weight at most U_j^a. So we have $|\sum \zeta_{P,i} X_i - R_P| \leq \sum_{i=1}^n U_i^a$ Now let $\chi(x) = x^a$. Then

$$\chi\left(\psi^{-1}(y) + c + 1\right) = \left((Ay)^{1/A} + c + 1\right)^a = \sum_{k=0}^{a} \binom{a}{k} (Ay)^{k/A} (c+1)^{a-k}$$

which is concave since the second derivative of each term of the sum is negative. The rest follows from Lemma 5.9. ∎

Now for this event *not* to occur, we must either have $\sum \kappa_i X_i \leq x\sqrt{K}$ or $\sum(\zeta_{P,i} - \kappa_i)X_i \leq \Delta - x\sqrt{K}$ for some P. But as the probability of a union of events never exceeds the sum of the probabilities of the events, the probability of failing in any of these ways is at most

$$\Pr\left[\sum \kappa_i X_i \leq x\sqrt{K}\right] + \sum_P \Pr\left[\sum(\zeta_{P,i} - \kappa_i)X_i \leq \Delta - x\sqrt{K}\right]$$
$$\leq \left[\Phi(x) + C_1\left(\frac{A^2}{n^{1/A}T_K}\right)^{1/5}\right] + n(p/n) \qquad (5.25)$$

by Lemmas 5.11 and 5.12. So the probability some processor decides 0 is at most (5.25), and thus the probability that all processors decide 1 is at least 1 minus (5.25). ∎

The running time of the protocol is more easily shown:

Theorem 5.15 *No processor executes more than $(AK)^{1/A}(2+n/c)+2c+2n$ register operations during an execution of the shared coin protocol.*

Proof: First consider the maximum number of votes a processor can cast. After $(AK)^{1/A}$ votes the total variance of the processor's votes will be

$$\sum_{x=1}^{(AK)^{1/A}} x^{2a} > \int_0^{(AK)^{1/A}} x^{2a}\,dx = \frac{\left((AK)^{1/A}\right)^A}{A} = K,$$

so after at most an additional c votes the processor will execute line 11 of Figure 5.1 and see a total variance greater than K. Thus each processor casts at most $(AK)^{1/A} + c$ votes. But each vote costs 1 write operation in line 8, and every c votes costs n reads in line 11, to which must be added a one-time cost of n reads in line 13. The total number of operations is thus at most $\left((AK)^{1/A} + c\right)(1 + \lceil n/c\rceil) + n \leq ((AK)^{1/A} + c)(2 + n/c) + n = (AK)^{1/A}(2 + n/c) + 2c + 2n$. ∎

It remains only to find values for a, K, and c which give both a constant agreement parameter and a reasonable running time. As a warm-up, let us consider what happens if we emulate the protocol of Bracha and Rachman [BR91]:

Theorem 5.16 *If $a = 0$, $K = 4n^2$, and $c = \frac{n}{4\log n} - 3$, then for n sufficiently large the protocol implements a shared coin with agreement parameter at least 0.05 in which each processor executes at most $O(n^2 \log n)$ operations.*

which for $n \geq 2$ will be less than $d\sqrt{T_K/nA} = 2$. To show (5.22), note that

$$g^A = \left(1 + \frac{1}{16 \log^2 n}\right)^{\log n} \leq e^{1/16 \log n}$$

and thus $\log(g^A) \leq 1/16 \log n$. But

$$\begin{aligned}
\log\left(1 + \frac{(x-d)^2}{2\log(n/p)}\right) &= \log\left(1 + \frac{1}{8\log(n/p)}\right) \\
&\geq \frac{1}{8\log(n/p)} - \frac{1}{128\log^2(n/p)} \\
&= \frac{1}{8(\log n - \log p)} - \frac{1}{128(\log n - \log p)^2}
\end{aligned}$$

(using the approximation $\log(1+x) \geq x - \frac{1}{2}x^2$). For sufficiently large n this quantity exceeds $1/16 \log n$ and (5.22) holds. The remaining constraint (5.23) is easily verified, and thus Theorem 5.14 applies and the agreement parameter is at least

$$\begin{aligned}
&1 - \left[\Phi(1) + C_1\left(\frac{\log^2 n}{n^{1/\log n}(16n\log n)}\right)^{1/5} + 1/10\right] \\
&\leq 1 - \left(0.842 + O\left((\log n/n)^{1/5}\right) + 0.10\right)
\end{aligned}$$

which is at least 0.05 for sufficiently large n. Thus the protocol gives a constant agreement parameter.

Now by Theorem 5.15, the number of operations executed by any single processor is at most $(AK)^{1/A}(2 + n/c) + 2c + 2n$, or

$$(\log n)^{1/\log n}(16n \log n)(n/\log n)^{1/\log n} O(\log n) + O(n)$$

which is $O(n \log^2 n)$. ∎

It follows immediately that plugging a coin with the parameters of Theorem 5.17 into the consensus protocol construction of Chapter 3 gives a consensus protocol that requires an expected $O(n \log^2 n)$ operations per processor. It is not difficult to see that the best bound we can place on the total number of operations is in fact n times this quantity, or $O(n^2 \log^2 n)$. The worst case is when each processor casts the same number of common votes.

	Expected operations	
	Per processor	Total
Abrahamson [Abr88]	$2^{O(n^2)}$	$2^{O(n^2)}$
Aspnes and Herlihy [AH90a]	$O(n^4)$	$O(n^4)$
Attiya, Dolev, and Shavit [ADS89]	$O(n^4)$	$O(n^4)$
Chapter 4 ([Asp90])	$O(n^2(p^2+n))$	$O(n^2(p^2+n))$
Bracha and Rachman [BR90]	$O(n(p^2+n))$	$O(n(p^2+n))$
Dwork et al. [DHPW92]	$O(n(p^2+n))$	$O(n(p^2+n))$
Saks, Shavit, and Woll [SSW91]	$O(n^4)$	$O(n^4)$
Bracha and Rachman [BR91]	$O(n^2 \log n)$	$O(n^2 \log n)$
Chapter 5 ([AW92])	$O(n \log^2 n)$	$O(n^2 \log^2 n)$

Table 6.1: Comparison of consensus protocols.

number of *active* processors as defined in Section 4.4. The first known protocol was the exponential protocol of Abrahamson [Abr88]. The first known polynomial-time protocol was that of Aspnes and Herlihy [AH90a]. Attiya, Dolev, and Shavit [ADS89] described a modification of this protocol which required only a bounded amount of space, but which retained the spirit of the rounds-based structure of the Aspnes-Herlihy protocol.

The protocol of Chapter 4, which also appears in [Asp90], was the first to eliminate the use of rounds by using a robust shared coin. Since its first appearance its performance was improved by a factor of n by Bracha and Rachman [BR90] and by Dwork et al. [DHPW92]. Both groups achieved the improvement by replacing the $O(n^2)$ implementation of an atomic counter with a weaker primitive that required only $O(n)$ register operations per counter operation, and acted sufficiently like a counter to make the consensus protocol work.

The first protocol to use the idea of casting votes until a quorum is reached (instead of until a sufficiently large margin of victory is reached) was that of Saks, Shavit, and Woll [SSW91]. Their protocol was optimized for the special case where nearly all of the processors are running in lockstep. Bracha and Rachman [BR91] noticed that the protocol could be sped up by having each processor read all the registers only after every $O(n/\log n)$ votes; the resulting protocol is a special case of the protocol of Chapter 5 obtained by setting a to 0. The protocol of Chapter 5, which also appears in [AW92], is the first to use votes of unequal weight, and as a result is the first for which

the per-processor bound.

However, to get below $\Omega(n^2)$ operations will require at least two breakthroughs. The first problem is that all of the algorithms we currently have require that every processor read every other processor's register directly at some point, which takes $\Theta(n^2)$ total operations. It seems likely that some sort of randomized cooperative technique could allow this dissemination of information to proceed more quickly (possibly at the cost of using very large registers); but at present no such technique is known.

The second problem is that to reduce the total number of operations below $\Omega(n^2)$ it will be necessary to reduce the number of local random choices below $\Omega(n^2)$, as local coin-flips that have no writes between them effectively consolidate into a single random choice from the point of view of the scheduler. This problem appears more difficult than the first, as it requires abandoning the voting technique at the heart of all currently known wait-free consensus protocols. The reason is that in these protocols, the scheduler's power only becomes limited when the standard deviation of the total vote becomes comparable to the sum of the votes that the scheduler can withhold. With unweighted votes, $\Omega(n^2)$ votes are required; for weighted votes the situation is only made worse, as increasing the weight of some votes increases the sum of the withheld votes more quickly than it increases the standard deviation of the total vote. It appears that it will be difficult to get below $\Omega(n^2)$ without adopting some decision method that takes more account of the ordering of events in the system.

6.3 Open problems

The consensus protocol described in Chapter 5 comes quite close to the limits of current methods for solving wait-free consensus. Aside from optimizations such as eliminating the $\log n$ factors from the per-processor bound or reducing the value of n at which the protocol becomes practical, essentially the only question remaining is whether the total number of operations can be reduced substantially. There are several questions whose answers would shed light on this problem, as well as many other problems in the area:

1. Is it possible in the asynchronous shared-memory model for n processors to collectively read n registers in fewer than $\Theta(n^2)$ total operations?

Bibliography

[AAD+90] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. In *Proceedings of the Ninth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 1–14, August 1990.

[Abr88] Karl Abrahamson. On achieving consensus using a shared memory. In *Proceedings of the Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 291–302, August 1988.

[ADS89] H. Attiya, D. Dolev, and N. Shavit. Bounded polynomial randomized consensus. In *Proceedings of the Eighth ACM Symposium on Principles of Distributed Computing*, pages 281–294. August 1989.

[AFL83] Eshrat Arjomandi, Michael J. Fischer, and Nancy A. Lynch. Efficiency of synchronous versus asynchronous distributed systems. *Journal of the ACM*, 30(3):449–456, July 1983.

[AG91] James H. Anderson and Bojan Groŝelj. Beyond atomic registers: Bounded wait-free implementations of non-trivial objects. In *Proceedings of the Fifth International Workshop on Distributed Algorithms*, pages 52–70. Springer-Verlag, 1991.

[AH90a] James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, September 1990.

ACM Symposium on Principles of Distributed Computing, pages 222–231, 1987.

[BR90] Gabi Bracha and Ophir Rachman. Approximated counters and randomized consensus. Technical Report 662, Technion, 1990.

[BR91] Gabi Bracha and Ophir Rachman. Randomized consensus in expected $O(n^2 \log n)$ operations. In *Proceedings of the Fifth International Workshop on Distributed Algorithms*. Springer-Verlag, 1991.

[BT83] Gabriel Bracha and Sam Toueg. Asynchronous consensus and byzantine protocols in faulty environments. Technical Report 83-559, Department of Computer Science, Cornell University, 1983.

[Che52] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23(4):493–407, December 1952.

[CIL87] Benny Chor, Amos Israeli, and Ming Li. On processor coordination using asynchronous hardware. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 86–97, 1987.

[CM89] Benny Chor and Lior Moscovici. Solvability in asynchronous environments. In *30th Annual Symposium on Foundations of Computer Science*, pages 422–427, October 1989.

[CMS89] Benny Chor, Michael Merritt, and David B. Shmoys. Simple constant-time consensus protocols in realistic failure models. *Journal of the ACM*, 36(3):591–614, July 1989.

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[DHPW92] Cynthia Dwork, Maurice Herlihy, Serge Plotkin, and Orli Waarts. Time-lapse snapshots. In *Proceedings of Israel Symposium on the Theory of Computing and Systems*, 1992.

[IL87] Amos Israeli and Ming Li. Bounded time stamps. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 371–382, 1987.

[Kop84] P.E. Kopp. *Martingales and Stochastic Integrals.* Cambridge University Press, 1984.

[LAA87] 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.

[Lam77] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[Lam86a] Leslie Lamport. On interprocess communication, part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[Lam86b] Leslie Lamport. On interprocess communication, part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[LF81] Nancy A. Lynch and Michael J. Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, 13:17–43, 1981.

[LL90] Leslie Lamport and Nancy Lynch. Distributed computing: Models and methods. In Jan Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter IX, pages 1157–1199. MIT Press, 1990.

[Lyn88] Nancy Lynch. I/O automata: A model for discrete event systems. Technical Report MIT/LCS/TM-351, MIT Laboratory for Computer Science, March 1988.

[NW87] Richard 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.