Bounded Polynomial Randomized Consensus

(PRELIMINARY VERSION)

Hagit Attiya* Danny Dolev† Nir Shavit‡

Abstract

In [A88], Abrahamson presented a solution to the randomized consensus problem of Chor, Israeli and Li [CIL87], without assuming the existence of an atomic coin flip operation. This elegant algorithm uses unbounded memory, and has expected exponential running time. In [AH89], Aspens and Herlihy provide a breakthrough polynomial-time algorithm. However, it too is based on the use of unbounded memory. In this paper, we present a solution to the randomized consensus problem, that is bounded in space and runs in polynomial expected time.

1 Introduction

The *Consensus Problem* in shared memory environment is that of providing an algorithm, by which n processes, running asynchronously and communicating via shared memory, can agree on a value. Loosely speaking, the algorithm should have the following properties:

1. *Consistency*: No two processes decide on different values;
2. *Validity*: If all processes have the same initial value, then processes decide on that value.
3. *Wait-freeness*: Each process is guaranteed to decide after a finite number of steps, independently of other processes.

In a shared memory in which only atomic read and write operations are allowed there is no deterministic solution to the problem. This result was directly proved by [AG88, CIL87, LA87] and implicitly can be deduced from [DDS87, FLP85]. Herlihy [H88] presents a comprehensive study of the problem, and of its implications on the construction of many synchronization primitives.

A *randomized* solution to the consensus problem is one in which, rather than being *guaranteed*, it is only *expected* that the number of steps until a process decides is finite, that is, property (3) above is replaced by:

3. *Finite expected waiting:* The expected number of steps until a process decides is finite.

Such an algorithm, provides a basis for constructing novel *universal* synchronization primitives, such as the *fetch and cons* of [H88], or the *sticky bits* of [P89].

[*] *MIT* Laboratory for Computer Science, supported by *NSF* contract no CCR-8611442, by *ONR* contract no N0014-85-K-0168, and by *DARPA* contract no N00014-83-K-0125

[†] *IBM* Almaden Research Center and Hebrew University, Jerusalem.

[‡] Hebrew University, Jerusalem. Supported by an Israeli Communications Ministry Award. Currently visiting the *TDS* group at *MIT*, supported by *NSF* contract no CCR-8611442, by *ONR* contract no N0014-85-K-0168, by *DARPA* contract no N00014-83-K-0125, and a special grant from *IBM*.

Chor, Israeli, and Li [CIL87] were the first to provide a time-efficient randomized solution to the problem, using bounded size memory. Their solution was based on the availability of a powerful *atomic coin flip* operation. In [A88], Abrahamson presented a first solution not assuming the existence of such an operation. However, this elegant algorithm uses unbounded memory, and has exponential expected running time. The question was thus raised:

> *Does there exist an algorithm that is polynomial in running time and bounded in memory size?*

An exponential time algorithm can be derived from that of [A88] (see [ADS89]) using a transformation based on the *concurrent time stamp system* techniques of [DS89]. Aspens and Herlihy (in [AH88]) provide a breakthrough algorithm that runs in polynomial expected time. Unfortunately, it is based on the use of unbounded size memory in a "stronger" way than in [A88]. Since for reasons presented in the sequel, there seems to be no transformation of [AH88] to a bounded protocol using concurrent time stamping techniques, the above question remained unanswered.

In this paper, we present a solution to the randomized consensus problem that both runs in polynomial expected time and is bounded in memory size.

The main reason for the simplicity in providing an exponential time randomized consensus algorithm using bounded space, is that all one need provide are actually the properties of *consistency* and *non-triviality*. The *wait-freeness*, i.e. exponential expected running time, is (though hard to analyze) just the result of the exponentially small probability that processes flipping independent coins, will come up with the same value. To provide the former two properties, one need only create a locking mechanism that will provide exclusion, before allowing processes to decide on a value. Such unbounded locking mechanisms are based on time stamping concurrent lock setting events, a process that has been shown to be modularly replaceable using bounded concurrent time-stamp systems.

In order to obtain an algorithm that runs in expected polynomial time, as [AH88], one must limit the ability of the adversary to create non-decision scenarios while processes try to lock for values. A way of doing this is by basing a process' decision to attempt to lock for a value, on a function of more than just one independent local coin toss, preferably on many coin tosses by all processes. This exact idea is abstracted into the notion of creating a *shared global coin* [CMS85]. Since attempts to lock for a value based on the shared coin could still fail (because as shown in [AH88], one cannot create a perfect coin) repeated global coin tosses are needed. When implementing multiple coin tosses, one must remember that processes run at different paces, so one should take care to *a.* prevent mixups between locations in memory used for new and old coins, and *b.* provide independence among shared coin flips (this means preventing processes in old coin toss phases, from causing attempts of processes in later coin tosses to fail). The algorithm uses an unbounded strip of coins, where for each toss a separate set of memory locations is allocated; this allows to distinguish between coin tosses, and thus to meet the above requirements.

Summing the above, in achieving polynomial expected time, unboundedness is used, not to order any two specific coin flipping events by *the relative times in which they occurred* (a property provided by concurrent time stamping), but by *how many coin flipping events is one process trailing behind the other.*

In [AH88], in addition to the above use of unbounded memory, the weak shared coin flip construction requires that each coin location in the unbounded strip be in itself unbounded. Finally, their use of a *random walk* to create the shared coin is based on a snapshot view of memory. The implementation of this snapshot operation also uses unbounded counters.

The main contribution of our paper is an implementation that achieves the properties of the coin strip using bounded memory. It is based on a technique for maintaining a "shrunken" version of the strip, effectively pulling together processes that opened a gap between one another. In addition, it is shown how to perform the random walk using only bounded coin locations. Finally, our algorithm is based on the availability of a memory primitive, on which a snapshot scan can be performed. We show how to implement such a primitive boundedly.

The rest of the paper is organized as follows. In *Section 2* a scannable memory primitive is defined and constructed. In *Section 3* a bounded memory implementation of a weak shared coin is presented. In *Section 4* the implementation of the coin strip is presented. We introduce a token game capturing the properties of the strip. A shrunken version of the game is shown to provide the same properties, and is then translated into a game on a weighted graph. Finally, a concurrent implementation of the game on the graph is presented. *Section 5* shows how bounded size strips of coins can be manipulated based on the concurrent graph game. All the unbounded constructs of the [AH88] type algorithm presented in *Section 5*, are then replaced by the bounded ones, providing the desired solution. In *Section 6*, an outline of the correctness proof of the algorithm is presented. Due to lack of space, some of the proofs are omitted.

2 Snapshot Scanning

2.1 Definitions

A *Scannable Memory* V is an abstract data type shared among n concurrent and completely asynchronous processes. There are two operations that any process can execute on V, a *write* operation and a *scan* operation. As discussed below, it is not assumed that these operations are necessarily *waitfree* [H88, AG88].

Assume that each process' program consists, among other, of the above two operations, whose execution generates a sequence of *elementary operation executions,* totally ordered by the *precedes* relation (of [L86a, L86c] denoted " $\longrightarrow$ "). The following

$$W_i^{[1]} \longrightarrow S_i^{[1]} \longrightarrow W_i^{[2]} \longrightarrow W_i^{[3]} \\ \longrightarrow S_i^{[2]} \longrightarrow S_i^{[3]} \longrightarrow S_i^{[4]} \longrightarrow \dots$$

is an example of such a sequence by process i, where $W_i^{[k]}$ denotes process i's k^{th} execution of a write operation, and $S_i^{[k]}$ the k^{th} execution of a scan operation (the superscript $[k]$ is used for notation, and is not visible to the processes). One should bear in mind that the asynchronous nature of the operations allows situations where a scan overlaps many consecutive write operations of other processes. Also, several consecutive scans could possibly be overlapped by a single write operation.

Let $\dashrightarrow$ be the *can affect* relation of [L86a, L86c]. A *global time model*[1] of operation executions is assumed (see [L86a, B88]). The following definition attempts to capture the notion that a possible effect of one operation on the shared memory (such as the writing of a value), existed at a point in global time where the other was being executed.

Definition 2.1. *A write operation execution $W_i^{[a]}$ potentially coexists with another operation execution $O_j^{[b]}$ (O stands for either a scan or write) if $W_i^{[a]} \dashrightarrow O_j^{[b]}$ and there does not exist a $W_i^{[a']}$ such that $W_i^{[a]} \longrightarrow W_i^{[a']} \longrightarrow O_j^{[b]}$.*

With each write operation execution $W_i^{[k]}$, a *value* $v_i^{[k]}$ written into V is associated. A *scan* operation returns a *view*, a set of values $\bar{v} = \{v_1^{[k_1]} \dots, v_n^{[k_n]}\}$[2].

The following requirement is made to assure that the snapshot view $\bar{v}$ returned by $S_j^{[b]}$ is a meaningful one, namely, returning the values of write events immediately before or concurrent with the scan, and not just any possible set of values.

P1 *regularity*: For any value $v_i^{[a]}$ in $\bar{v}$ of $S_j^{[b]}$, $W_i^{[a]}$ potentially coexisted with $S_j^{[b]}$.

The above eliminates uninteresting trivial solutions and introduces a measure of liveness into the system. More importantly, it implies that the behavior of the scannable memory is as if it consists of disjoint registers, one per process, which the designated process can write, and all can read. This is very different from the behavior of multi reader multi writer atomic registers, where the latest write of any process erases the values written by others.

[1] Implying that for any two operation executions, $a \longrightarrow b$ or $b \dashrightarrow a$.

[2] Initialization and safety are similar to *Axioms B0-3* for single-writer *atomic registers* [L86b]

Though a scan as above is sufficient for many applications, one is interested in a scan that returns an "instantaneous" view of memory, that is, having the following stronger property:

P2 *snapshot*: For any two values $v_i^{[a]}$ and $v_j^{[b]}$ in $\bar{v}$ of $S_k^{[c]}$, $W_i^{[a]}$ potentially coexisted with $W_j^{[b]}$, or $W_j^{[b]}$ potentially coexisted with $W_i^{[a]}$, or both.

Though *P1-2* return values that could have been returned by an instantaneous scan, they do not imply that scan operations of all processes are serializable. Moreover, they do not imply that later scans will obtain later snapshot views. The following property is therefore added, to formalize, together with *P1-2*, the idea that all scans are serializable.

P3 *scan serializability:* Let $S_j^{[b]}$ and $S_k^{[b']}$ be any pair of scans. Let $v_i^{[a_i]}$ and $v_i^{[a'_i]}$, $i \in \{1..n\}$, denote the corresponding values returned by the two scans. Then either for every $i \in \{1..n\}$, $a_i \leq a'_i$, or for every $i \in \{1..n\}$, $a'_i \leq a_i$.

For the purposes of the applications in this paper, it is not required that both *scan* and *write* operations be *waitfree* [H88, AG88]. Since every process' execution sequence will be an alternating sequence of *scan* followed by *write*, it will actually suffice that in any infinite system execution, there exists a new write operation infinitely often. In the full paper, a formal treatment of this property is provided.

2.2 Bounded Implementation of Scannable Memory

The implementation is based on the use of *single-writer-multi-reader* and *two-writer-two-reader* atomic registers. The *scannable memory V* will consist of *n single-writer-multi-reader* atomic registers V_i, $i \in \{1..n\}$, each V_i written by process i and read by all. In addition, for every pair of processes i and j, a pair of *two-writer-two-reader* atomic registers A_{ij} and A_{ji} are maintained [3]. Bounded constructions of such registers from weaker primitives are shown in [Bl87, L86b, IL88, BP87, N87, SAG87, LV88, DS89]. Register A_{ij} is used by i to inform j that it has updated V_i, and by j to mark that it has read V_i. To simplify the proofs (and only for this purpose), an alternating bit field is assumed to be added to each register V_i, such that two values written in consecutive writes by the same process, always differ.

The main idea behind the implementation of the *scan* and *write* operations is as follows. A value of 1 in register A_{ji} denotes an "arrow" pointing from j to i, a value of 0 denotes an arrow from i to j. To scan the memory, a process i will direct all arrows A_{ji} towards other processes, perform a collecting of values followed by a collecting of arrows, and repeat these two collections again. If the values have not changed and no arrow has been redirected towards it, process i has collected a snapshot in its second read of every register. [4] To write a value, a process j directs the arrows A_{ji} towards any possibly-scanning process, notifying that it has started a write, then writes the value. The following are the *write* and *scan* procedures of a process i, where we use the notation $j \in \{1..n\} - \{i\}$ to denote that indexing is performed in some arbitrary order.

```
procedure write(value);
  begin
    for $j \in \{1..n\} - \{i\}$ do $A_{ij} := 1$ od;
    $V_i := value$;
  end write;
```

Assume that a process, during the execution of the scan operation, has seen no arrows redirected, and both values being the same. It can thus deduce that no process whose corresponding value it returns, could have performed its following write, completely before any of the other writes whose values it returns. The reason is that if that were the case, the writing process would have turned the arrow and the scan would have gone through another round.

```
function scan
  begin
```

[3] To save in the complexity of constructing multi writer registers, the *arrows* technique of [DGS88] can be used.

[4] The two phases of *value*-collecting are also used to simplify the proofs.

```
L:  for $j \in \{1..n\} - \{i\}$ do $A_{ji} := 0$      od;
    for $j \in \{1..n\} - \{i\}$ do $V1[j] := V_j$     od;
    for $j \in \{1..n\} - \{i\}$ do $V2[j] := V_j$     od;
    for $j \in \{1..n\} - \{i\}$ do $A[j] := A_{ji}$    od;
    if $(\exists j)(A[j] = 1 \vee V1[j] \neq V2[j])$
        then goto L fi;
    return $V2$;
  end scan;
```

Though the *write* operation is waitfree, the *scan* operation is of course not, because scans may repeatedly be forced to return to *line* L. However, scans do not wait for other scans, and the above can only happen on account of repeated execution of new write operations by some process. Thus, it can be proven that the implementation provides the type of progress described in the previous section.

The following is the main core of the proofs of properties *P1-3*. The notation $r1_k^{[b]}(V_{ij})$ for example, will denote the first read in scan operation execution $S_k^{[b]}$ of register V_{ij}.

Lemma 2.1. *For any value $v_i^{[a]}$ in $\overline{v}$ of $S_j^{[b]}$, $W_i^{[a]}$ potentially coexisted with $S_j^{[b]}$.*

Proof Assume by way of contradiction that the claim does not hold. There must thus exist some value $v_i^{[a]}$ in $\overline{v}$ of $S_j^{[b]}$, such that $\neg(W_i^{[a]} \dashrightarrow S_j^{[b]})$ or $(\exists W_i^{[a']})(W_i^{[a]} \longrightarrow W_i^{[a']} \longrightarrow S_j^{[b]})$. By the assumption of global time, $\neg(W_i^{[a]} \dashrightarrow S_j^{[b]})$ implies $S_j^{[b]} \longrightarrow W_i^{[a]}$, which by atomic register axiom *B4* of [L86c], it cannot be that $v_i^{[a]}$ was returned. Thus, the second condition must hold, which by the scan algorithm implies

$$w_i^{[a]}(V_i) \longrightarrow w_i^{[a']}(V_i) \longrightarrow r2_j^{[b]}(V_i)$$

where $v_i^{[a]}$ was returned in $r2_j^{[b]}(V_i)$, a contradiction to atomic register axiom *B4* of [L86c]. ■

This implies *P1*, the following proves *P2* is met.

Lemma 2.2. *For any two values $v_i^{[a]}$ and $v_j^{[b]}$ in $\overline{v}$ of $S_k^{[c]}$, $W_i^{[a]}$ potentially coexisted with $W_j^{[b]}$ or $W_j^{[b]}$ potentially coexisted with $W_i^{[a]}$ or both.*

Proof Assume by way of contradiction that the claim does not hold. There must thus exist two values $v_i^{[a]}$ and $v_j^{[b]}$ in $\overline{v}$ of $S_k^{[c]}$, such that neither $W_j^{[b]}$, nor $W_i^{[a]}$, potentially coexisted with the other. W.l.o.g, it must be that

$$(\exists W_i^{[a']})(W_i^{[a]} \longrightarrow W_i^{[a']} \longrightarrow W_j^{[b]}).$$

By the scan algorithm, $w_k^{[c]}(A_{jk}) \longrightarrow r_k^{[c]}(V_i)$. Since $v_i^{[a]}$ and not $v_i^{[a']}$ was returned in $r_k^{[c]}(V_i)$, $r_k^{[c]}(V_i) \longrightarrow w_i^{[a']}(V_i)$. Because $W_i^{[a']} \longrightarrow W_j^{[b]}$, it must be that $w_i^{[a']}(V_i) \longrightarrow w_j^{[b]}(A_{jk}) \longrightarrow w_j^{[b]}(V_j)$. Also, because $v_j^{[b]}$ was returned in $r_k^{[c]}(V_j)$, it is must be the case that $w_j^{[b]}(V_j) \longrightarrow r_k^{[c]}(V_j)$. Again by the scan algorithm, $r_k^{[c]}(V_j) \longrightarrow r_k^{[c]}(A_{jk})$. From the above, by the transitivity of $\longrightarrow$, it follows that

$$w_k^{[c]}(A_{jk}) \longrightarrow w_j^{[b]}(A_{jk}) \longrightarrow r_k^{[c]}(A_{jk}).$$

Since in $w_j^{[b]}(A_{jk})$ a value of 0 was written, this value must have been read in $r_k^{[c]}(A_{jk})$, a contradiction to the termination condition of the scan algorithm. ■

Using similar arguments the next two lemmas prove *P3*. The following lemma establishes that in the two reads of any scan operation execution, the value written in the exact same write is returned.

Lemma 2.3. *In any scan operation execution $S_k^{[c]}$, for any value $v_i^{[a]}$ in $\overline{v}^{[c]}$, $v_i^{[a]}$ was read in both $r1_k^{[c]}$ and $r2_k^{[c]}$.*

Proof Assume by way of contradiction that the above does not hold. Since the values read in $r1_k^{[c]}$ and $r2_k^{[c]}$ must be the same, and two consecutive writes have different toggle bit values, it must be that for $v_i^{[a'']}$ and $v_i^{[a]}$ returned in $r1_k^{[c]}$ and $r2_k^{[c]}$ respectively, there must exist a write operation execution $W_i^{[a']}$ such that

$$W_i^{[a'']} \longrightarrow W_i^{[a']} \longrightarrow W_i^{[a]}.$$

In a manner similar to that of the former proof, by the ordering of reads of A_{ik} and V_i, it must be that

$$w_k^{[c]}(A_{ik}) \longrightarrow r1_k^{[c]}(V_i) \longrightarrow w_i^{[a']}(V_i) \longrightarrow w_i^{[a]}(A_{ik})$$

$$\longrightarrow w_i^{[a]}(V_i) \longrightarrow r2_k^{[c]}(V_i) \longrightarrow r_k^{[c]}(A_{ik}).$$

This implies that the value of 0 written in $w_i^{[a]}(A_{ik})$ must have been read in $r_k^{[c]}(A_{ik})$, a contradiction to the scans termination condition. ■

Lemma 2.4. *Let $S_x^{[c]}$ and $S_y^{[c']}$ be any pair of scans. Let $v_i^{[a_i]}$ and $v_i^{[a'_i]}$, $i \in \{1..n\}$, denote the corresponding values returned by the two scans. Then either for every $i \in \{1..n\}$, $a_i \leq a'_i$, or for every $i \in \{1..n\}$, $a'_i \leq a_i$.*

Proof Assume by way of contradiction that the claim does not hold. There must thus exist values $v_i^{[a]}$ and $v_j^{[b]}$ in $\overline{v}^{[c]}$, and $v_i^{[a']}$ and $v_j^{[b']}$ in $\overline{v}^{[c']}$ such that $a < a'$ and $b > b'$.

Lemma 2.3 implies that the value returned in both reads of a scan operation execution is of the same write operation. In the scan operation execution of y, Since in $r1_y^{[c']}(V_i)$, $v_i^{[a']}$ was returned, $w_i^{[a']}(V_i) \longrightarrow r1_y^{[c']}(V_i)$. Since in $r2_y^{[c']}(V_j)$, $v_j^{[b]}$ was not returned, $r2_y^{[c']}(V_j) \longrightarrow w_j^{[b]}(V_j)$. By the order of reads in a scan it thus follows that

$$w_i^{[a']}(V_i) \longrightarrow r1_y^{[c']}(V_i)$$
$$\longrightarrow r2_y^{[c']}(V_j) \longrightarrow w_j^{[b]}(V_j).$$

By similar arguments, regarding the scan operation execution of x,

$$w_j^{[b]}(V_j) \longrightarrow r1_x^{[c]}(V_j)$$
$$\longrightarrow r2_x^{[c]}(V_i) \longrightarrow w_i^{[a']}(V_i).$$

By transitivity, the combination of these two sequences of operation executions contradicts the antisymmetry property of the partial order $\longrightarrow$.
■

3 A Bounded Implementation of a Shared Coin

The implementation of the weak shared coin is based on the *random walk* technique of [AH88]. For lack of space we explain only the modification allowing to bound the size of the counters used to implement the coin. The main idea of the modification used is rather straightforward. The coin implemented by the random walk is *weak*, that is, involves a small probability that processes will disagree on the coin's outcome. Thus, one can allow a process to always decide *heads* in case its counter overflows, as long as the probability of this event can be absorbed into the probability of processes disagreeing on the outcome.

Let $\bar{c} =< c_1, \ldots, c_n >$ be an array of counters implementing a shared coin. Each counter c_i has values in the range $\{-(m+1)..(m+1)\}$, written by its corresponding process i. Let $walk\_value(\bar{c}) = \sum_{i=1}^{1=n} c_i$. The following are thus the functions of process i, for determining if the random walk has led to a coin value, and for performing a step in the random walk by process i.

```
function coin_value ($\bar{c}$);
  begin
    if $c_i \notin \{-m..m\}$ then
        return heads fi;
    if $walk\_value(\bar{c}) > \delta \cdot n$ then
        return heads
        elseif $walk\_value(\bar{c}) < -\delta \cdot n$ then
        return tails
            else return undecided fi fi;
  end coin_value;

procedure walk_step;
   begin
     if flip = heads then $c_i := c_i + 1$
                     else $c_i := c_i - 1$ fi;
   end walk_step;
```

Lemma 3.1 (Aspnes and Herlihy). *The probability that two processes will disagree on the coins outcome is $(\delta - 1)/(2\delta)$.*

Lemma 3.2 (Aspnes and Herlihy). *The expected number of steps until the coin is decided is $(\delta + 1)^2 n^2$.*

Look at a random walk starting from 0 with barriers at b and $-b$, consisting of the steps:

$$\delta_1, \delta_2, \ldots \qquad \delta_i \in \{-1, +1\} \text{ for all } i.$$

The following is a bound on the *probability* that after m steps, none of the barriers was crossed. Define

$$S_m = \textbf{Prob}\left[|\sum_{i=1}^{m} \delta_i| \leq b \right]$$

Clearly, the desired probability is bounded from above by S_m. Thus,

Lemma 3.3. *Let $m = (f(b)b)^2$, for some function f, then there exists a constant C, such that $S_m \leq \frac{C}{f(b)}$* (proof ommited).

Based on the above, one can prove that by choosing m to be large enough, the probability that the adversary can force processes to disagree because of the deterministic choice of *heads* in case of counter overflow, is negligible, as formalized by the following lemma:

Lemma 3.4. *There exists a constant C such that the probability that in the random walk generated by a sequence of executions of the algorithm on a given coin $\bar{c}$,*

$$\textbf{Prob}\ [|c_i| \geq m] \leq \frac{C \cdot \delta \cdot n}{\sqrt{m}}.$$

4 The Rounds Strip

In this section a method is shown for replacing the unbounded strip of round locations required by the algorithm of [AH88], by a bounded construct. The important observation is that this algorithm utilizes the rounds strip in a very restricted way. Informally

Observation 1. *There exists a constant K such that at any point in the computation:*

1. *The actions performed by any process are not affected by values of processes that are strictly more than K rounds behind it.*
2. *If a process performs round r, and cannot decide, then there is a disagreement about the value of the shared coin of round $r - K$. This implies that when this process proceeds to round $r + 1$, it can withdraw its contribution to the coin of round $r - K$, without affecting the performance of the algorithm.*

Thus, a complete picture of the rounds in which processors are located is not necessary; rather, it suffices to maintain a "compressed" description of the *distances* between these round numbers, and to save processes' contributions to the K latest coins that were flipped. The following subsections present the data structure used to maintain these distances concurrently.

In the next subsection, a simple game is presented in order to make precise the notion of "compression" mentioned above. Then, in *Section 4.2*, we show how to store and play this game using a *directed weighted graph*. In order to simplify the presentation this game is *sequential*. In *Section 4.3*, a data structure that implements the game on the graph is defined, as well as the procedures for playing the game on this graph *concurrently*.

The main problem is how to maintain the relevant values using bounded space, given that processes are asynchronous. For example, it could be that process will start flipping a coin in a round r when round r is maximal, and during its coin flipping other processes will move to higher rounds, that are an unbounded number of coin flips ahead.

4.1 The Game

Imagine the changes to the processes' round numbers as a game played on the natural numbers (viewed as an infinite ordered set of points):

Each processor controls a *token*, placed at a specific point, initially 0. Denote by r_i the location of i's token. Each processor can perform the step $move\_token_i$ that places its token at place $r_i + 1$. The game is a (possibly infinite) sequence of the form $move\_token_{i_1}$, $move\_token_{i_2}$ ...

At any stage of the game, the collection of tokens' positions forms a multi-set of integers, $S = \{r_1, \ldots, r_n\}$. Let π be the ordering permutation of S, i.e., $S = \{r_{\pi(1)} \leq r_{\pi(2)} \leq \cdots \leq r_{\pi(n)}\}$. Let K be some fixed constant. We now introduce two transformations, that, when applied to the set S, produce a "compressed" representation of it, without losing important information.

Shrinking. One is interested in the exact distance between two token if and only if, the distance between them is less than K. The goal of the first transformation is to "shrink" gaps of length strictly larger than K, to be of size K. Informally, $shrink_K(S)$ is a new set S', in which $r_{\pi(n)}$ remains in its current position, while any two consecutive tokens ($r_{\pi(i)}$ and $r_{\pi(i+1)}$) that are more than K apart, become K apart, while the distance between tokens that are less than K apart, remain unchanged.

Formally, let $S = \{r_{\pi(1)} \leq \cdots \leq r_{\pi(n)}\}$. Let $gap_i = r_{\pi(i)} - r_{\pi(i+1)}$, for $1 \leq i < n$, and define $shrink_K(S) = \{r'_{\pi(1)} \leq \cdots \leq r'_{\pi(n)}\}$, (for some parameter K) inductively as follows:

(1) $r'_{\pi(1)} = r_{\pi(1)}$.

(2) Assume we have defined $r'_{\pi(i)}$, then

$$r'_{\pi(i+1)} = \begin{cases} r'_{\pi(i)} + K & \text{if } gap_i > K \\ r'_{\pi(i)} + gap_i & \text{otherwise} \end{cases}$$

Intuitively, any "gap" in the sequence, whose length is strictly larger than K, is "shrunk" to be of length exactly K.

The *shrunken token game* is conducted by executing a $shrink_K$ on the set of token places after each $move\_token_{i_\ell}$ step, before the next $move\_token_{i_{\ell+1}}$ step.

Normalizing. It is easy to see that after applying $shrink_K$ to any set S, the distance between the maximal element and the minimal element is at most $K{\cdot}n$. To compress the values even further they are normalized, so that all values remain in a bounded range.

The ordering permutation of $S' = shrink_K(S)$ is still π. The transformation $normalize_K(S')$ maps each element $r'_i \in S'$ to $(r'_i - r_{\pi(n)}) + K \cdot n$. That is, the maximal token(s) is positioned at $K \cdot n$, and the rest of the tokens are move behind it while maintaining the distances between tokens. Notice that for any set S, all the values in $normalize_K(shrink_K(S))$ are in the range $[0..K{\cdot}n]$.

The *normalized shrunken game,* is conducted by applying $shrink_K$ and then $normalize_K$ to the set of token places after each $move\_token_{i_\ell}$ step, before the next $move\_token_{i_{\ell+1}}$ step.

An important property preserved by the normalized shrunken game is:

Non-Passive Shrinking. For any two token positions r_i and r_j in a state of the game, s.t. $0 \leq r_i - r_j \leq K$, if for later token positions, r'_i and r'_j, we have $r'_i - r'_j = (r_i - r_j) - 1$, then there is a $move\_token_j$ between the two states.

4.2 Representation as a Finite Graph

Given a state S of the above game, we define its *distance graph* $G(S)$, as follows: G is a directed weighted graph with nodes $V = \{1..n\}$, corresponding to tokens, one per process, edges $E = \{(i,j) \mid r_j \leq r_i\}$ indicating relative order of token locations, and weights $w(i,j)$, defined for any $(i,j) \in E$ as

$$w(i,j) = \begin{cases} r_i - r_j & \text{if } r_i - r_j \leq K \\ K & \text{otherwise.} \end{cases}$$

The following properties of the distance graph G, are implied from the definition of the normalized shrunken token game:

1. For any i and j in V, at least one of (i,j) or (j,i) is in E; both edges are in E if and only if the weight of both is 0.
2. There is no positive cycle, that is, a cycle including an edge (i,j) with $w(i,j) > 0$.
3. Let $P(i,j)$ be the set of all directed simple paths from i to j. For every path $\varphi \in P(i,j)$, let $W(\varphi) = \sum_{(u,v)\in\varphi} w(u,v)$. It follows from the above properties that $0 \leq W(\varphi) \leq K \cdot n$.
4. For any two directed paths φ_1 and $\varphi_2 \in P(i,j)$, either $W(\varphi_1) = W(\varphi_2)$, or there exists an edge $(u,v) \in \varphi_1$ such that $w(u,v) = K$.
5. For any i and j, such that $P(i,j) \neq \emptyset$, define

   $$dist(i,j) = \max_{\varphi' \in P(i,j)} (W(\varphi'))\},$$

   and define $max\_paths(i,j)$ to be

   $$\{\varphi \in P(i,j) \mid W(\varphi) = dist(i,j)\}$$

   Then $W(\varphi) = r_j - r_i$ for every $\varphi \in max\_paths(i,j)$.

Let $inc(i,G)$ be defined as the following transformation of graph G for a given i:

```
for all $j \neq i$ in $V$ do
  if $(j,i) \in G$ and
     $(\exists k)((j,i) \in max\_paths(k,i))$ then
     $w(j,i) := w(j,i) - 1$ fi;
  if $(i,j) \in G$ and
```

```
        $0 < w(i,j) < K$ then
            $w(i,j) := w(i,j) + 1$ fi;
    if $w(j,i) < 0$ then
        $E := E - \{(j,i)\} \cup \{(i,j)\}$;
        $w(i,j) = -w(j,i)$ fi;
  od;
```

Claim 4.1. *For a state S' reached from state S by a token_move of i in token game A,* $G(S') \equiv$ *inc*$(i, G(S))$.

4.3 Implementation of the Graph

Property (1) of the distance graph implies that the weights of all (undirected) edges suffice to induce the directed graph structure. The weights are maintained in a collection of $e_i[1..n]$ of *edge counters*, one per each (undirected) edge ($e_i[i]$ is not used). Each pair $e_i[j]$ and $e_j[i]$ of counters in the range $\{0..3{\cdot}K-1\}$, represents two pointers (of i and j, respectively) to a cycle of size $3{\cdot}K$. By incrementing the counter, a process moves its pointer a in clockwise direction (all arithmetics in this subsection is modulo $3{\cdot}K$).

Assume $e_i[j] - e_j[i] \leq e_j[i] - e_i[j]$ then the edge is (i,j), and $w(i,j) = e_i[j] - e_j[i]$, and vice versa. Thus, given two edge counters $e_i[j]$ and $e_j[i]$, the existence of a given directed edge is determined by the rule

$$(i,j) \in G \text{ if } (e_i[j] - e_j[i]) \leq (e_j[i] - e_i[j])$$

and the weight $w(i,j)$ of the edge (i,j) is $(e_i[j] - e_j[i])$. Note that if $e_i[j] = e_j[i]$, then we have both edges, (i,j) and (j,i) with both weights equal to 0. To keep the weight $w(i,j)$ in the range $\{0..K\}$, a process i does not increment $e_i[j]$ unless it is the trailing pointer, or it leads by less than K.

Let *make_graph* be the procedure that, given the collection of all edge counters, creates a graph representation, as described above. The following procedure is thus the (possibly concurrent) implementation of one increment move on the graph G.

```
function inc_graph($e_1[1..n]..e_n[1..n]$);
  begin
    $G :=$ make_graph($e_1[1..n]..e_n[1..n]$);
    for $j := 1$ to $n$ skip $i$ do
      if (($(j,i) \in G$ and
          $(\exists k)((j,i) \in max\_paths(k,i))$) or
          ($(i,j) \in G$ and $w(i,j) < K$) then
          $e_i[j] := e_i[j] + 1 \bmod 3K$
      fi;
    od;
  end;
```

5 The Algorithm

Based on Observation 1 (*Section 4*), if a process advanced K rounds ahead of another, it can erase its contribution to the trailing process' coin. A trailing process performing *next_coin_value* using that location will possibly see that process' counter as 0, but this can only cause it to perform an additional expected $O(n^2)$ steps (by Lemma 3.2), before advancing to the next round[5].

The *round* field of any value w_i consists of two fields: *coin* and *edge_counters*. The *coin* field is an array of *coin counters* $c_i[\alpha], \alpha \in \{0..K\}$, with an added *current_coin* pointer in the range $\{0..K\}$[6]. These counters are used to maintain the local parts of coins corresponding to the latest K rounds executed by process i. The counter to be used for the next coin of process i is determined by the function *next*(*current_coin*$_i$), returning *current_coin*$_i$ mod $(K + 1)$. The edge counters field is an array of n edge counters as described in Subsection 4.3. Initially all the above are 0. The following is thus the bounded implementation of the coin flipping and round incrementing operations for process i.

```
function next_coin_value(round);
    begin
      $G :=$ make_graph($e_1[1..n]..e_n[1..n]$);
      $\bar{c}[i] := coin_i[next(current\_coin_i)]$;
      for $j := 1$ to $n$ skip $i$ do
          if $(j,i) \in G$ and $w(j,i) \leq K$ then
              $\bar{c}[j] := coin_j[(current\_coin_j -$
                    $w(j,i) + 1) \bmod (K + 1)]$
            else $\bar{c}[j] := 0$ fi od;
      return coin_value($\bar{c}$);
    end;
```

[5] Several modifications that will improve the expected running time here and elsewhere in the algorithm are possible, but are not introduced for the sake of simplicity.

[6] In the procedures below, all fields are first written to a local variable, on which the *write* operation of the scannable memory is then performed.

```
procedure flip_next_coin(round);
    begin
        walk_step (coin_i[next (current_coin_i)]);
    end;

function inc(round);
    begin
        current_coin_i := next(current_coin_i);
        coin_i[next(current_coin_i)] := 0;
        inc_graph(e_1[1..n], .., e_n[1..n]);
    end;
```

In the above procedure, note that a process prepares, when advancing to a new round, the coin counter for flipping the coin in the next round.

We assume that processors start with binary initial values; however, the protocol can be extended to handle arbitrary initial values. Let K be 2, the following is thus the consensus algorithm for processor i, with initial value v_i. Process i is a *leader* if for all $j \neq i$, (i, j) is in G, that is having r_i equal to or dominating all other r_j. Process i agrees with process j, if both prefer the same value $v \neq \perp$.

```
write([pref: v_i, round: inc(round)])
repeat forever
1: scan;
2: if all who disagree
      trail by K and I'm a leader
      then decide(pref);
3: elseif leaders agree then
4:    write([pref: v, round: inc(round)])
5: elseif pref ≠ ⊥ then
6:    write([pref: ⊥, round: round])
   elseif next_coin_value(round) =
      undecided then
7:    write([pref: ⊥,
              round: flip_next_coin(round)])
   else
8:    write([pref: next_coin_value(round),
              round: inc(round)])
   fi fi fi fi;
end;
```

6 Proof of Correctness

The following section outlines the proofs that the algorithm has the properties of *consistency*, validity, and that it terminates in *polynomial expected time*. To simplify the proofs, the notion of a *virtual global round* is introduced, supporting the illusion that a process has an unbounded and monotonically non-decreasing round number, and that a unique shared coin is associated with each round.

6.1 Virtual Global Rounds

The serializability property (*P3*) of scan operation executions, implies that there is some linear ordering on the scan operation executions performed by all processes. Throughout the proof, let $S^{\{a\}}$ denote the a^{th} scan in this ordering, if the a^{th} scan is performed by process j, denote it by $S_j^{\{a\}}$. One scan operation execution is said to be *later* than another, if it is greater in this ordering. In the consensus protocol processes alternate between performing write and scan operations. This implies that between any two scans, $S^{\{a\}}$ and $S^{\{a+1\}}$, there is at most one write by any process. Denote by $var^{\{a\}}$ the value of any variable *var* that was read by $S^{\{a\}}$.

With each process i, in the a^{th} scan, a *virtual global round* is associated, denoted by $round(i, S^{\{a\}})$. The definition is by induction on the ordering among scan operation executions.

Base case. For all i, $round(i, S^{\{1\}}) = 0$.

Inductive step. Given $round(i, S^{[a-1]})$, let

$$max = \max_{i \in \{1..n\}} round(i, S^{\{a-1\}}),$$

$$old\_leaders(S^{\{a-1\}}) = \{j \mid round(i, S^{\{a-1\}}) = max\},$$

and

$$new\_leaders(S^{\{a\}}) = \{j \mid j \in old\_leaders(S^{\{a-1\}}) \text{ and } e_j[1..n]^{\{a\}}(j) \neq e_j[1..n]^{\{a-1\}}(j)\}.$$

Based on the above definitions, define $round(i, S^{\{a\}})$ as follows. If $new\_leaders(S^{\{a\}}) \neq \emptyset$, let $j^* \in new\_leaders(S^{\{a\}})$ and define

$$round(i, S^{\{a\}}) = \begin{cases} max+1 & i \in new\_leaders(S^{\{a\}}) \\ max+1 - dist(i, j^*) & \text{otherwise.} \end{cases}$$

In case the set $new\_leaders(S^{\{a\}}) = \emptyset$, let $j^* \in old\_leaders(S^{\{a\}})$ and define

$$round(i, S^{\{a\}}) = max - dist(i, j^*).$$

The above definition is simply that if one of the leaders in the former scan operation execution moved, all new processes are ordered relative to it, and otherwise they are ordered relative to the old leaders. Note that though the virtual global round of a process might change even without its performing an *inc* operation, it can only increase, that is, the virtual global round is a non-decreasing function.

In the following subsections, a *round* means a *virtual global round* unless otherwise stated. A process p is said to be in round r, starting from the first scan operation execution in which it was returned as being in r (determined by applying the above definition), and in all later scan operation executions until it is returned as being in a round $r' > r$. A round is said to be among the K largest (for some constant K) starting from the earliest scan operation execution in which some process is in this round and no other process is in a round greater by K, and until the first later scan operation execution for which there is a process in a round greater by K.

6.2 Consistency and Validity

Though we have attempted to maintain the general structure of the correctness and complexity proofs for the unbounded implementation of [AH88], by introducing virtual global rounds, the differences between our rounds strip implementation and the infinite rounds strip used in [AH88], force us to modify some of the statements, and to change most of the proofs.

For simplicity, it is assumed that there are only two possible input values, where $\bar{v}$ denotes the value different from v, for $v \in \{0, 1\}$. A process p *prefers* v in round r, if for some scan $S^{\{a\}}$ it is the case that $round(p, S^{\{a\}}) = r$, and $pref_p^{\{a\}} = v$. We have

Lemma 6.1. *If process p prefers v in round r and prefers $\bar{v}$ in round $r' > r$, then some process $q \neq p$ preferred $\bar{v}$ in round $r'' \geq r$.*

Proof (Sketch) By the algorithm, a process changes its preference only by executing *inc*. Let $S_p^{\{a\}}$ be the scan performed by p before executing this *inc*. This can occur only if some other process, say q, had $pref_q^{\{a\}} = \bar{v}$, and that in the graph returned in $S_p^{\{a\}}$, q has non-negative distance from p. Since rounds are monotonically non-decreasing, it is the case that $round(q, S_p^{\{a\}}) \geq round(p, S_p^{\{a\}})$ and the claim follows. ■

The above lemma and the code of the algorithm implies the following two lemmas.

Lemma 6.2. *If no process prefers $\bar{v}$ at round r when round r is among the 2 largest rounds, then no process prefers $\bar{v}$ at any round $r' > r$.*

Lemma 6.3. *If no process prefers $\bar{v}$ at round r when round r is among the 2 largest rounds, then no process is busy in any round $r' > r$.*

Lemma 6.4. *If every process that completed round r, when round r was among the 2 largest rounds, preferred v in round r, then every non-faulty process decides v by round $r + 1$.*

Lemma 6.4 implies *validity*, since if all processes start with the same input value they all prefer this value in round 1. Hence all processes will halt at round 2.

Lemma 6.5. *If any process decides in round r, then no process will ever be in a round larger than $r + 2$.*

The above lemma implies that all processes will execute round r when it is among the 2 largest rounds. We use this fact to prove that the algorithm has the *consistency* property.

Lemma 6.6. *If some process decides in round r then all processes will decide on the same value by round $r + 1$.*

6.3 Expected Running Time

A process is said to have selected its preference for round r *deterministically*, if it executed the corresponding *inc* in line 6. Similarly, a processor is said to have selected its preference for round *r randomly*, if it executed the corresponding *inc* in line 10. The following lemma assures that all processors that select their preference deterministically, select the same value.

Lemma 6.7. *If processes p and q deterministically selected v and v′, respectively, as their preferences for round r, when r was among the* 2 *largest rounds, then* $v = \bar{v}$.

Hence, one may talk about the deterministic value preferred in a certain round. The next lemma shows that the scheduler is forced to decide on the deterministic value of a round before any process starts flipping a coin for that round.

Lemma 6.8. *If process p is deterministic in round r, and process q is randomized in round r, then p wrote its preference for round r before q started to perform flip_next_coin.*

This lemma implies that decisions in different rounds are independent events. Thus, the probability of deciding in any round is that of a sequence of independent Bernoulli trials, with success probability ϵ, for some constant $\epsilon > 0$ (this follows from Lemmas 3.1 and 3.4). Hence the expected number of rounds executed before the algorithm terminates is constant. As each shared coin is flipped in polynomial expected number of steps (Lemma 3.2), the algorithm terminates in a polynomial expected number of steps.

Acknowledgements. The authors wish to thank Yehuda Afek and Michael Merritt for observations regarding scannable memory, made in the course of ongoing research. Thanks are also due to Roy Meshulam for helpful discussions.

References

[A88] K. Abrahamson, "On Achieving Consensus Using a Shared Memory," *Proc. 7th ACM Symp. on Principles of Distributed Computing*, 1988, pp. 291-302.

[AG88] J. H. Anderson, and M. G. Gauda, "The Virtue of Patience: Concurrent Programming With and Without Waiting," unpublished manuscript, Dept. of Computer Science, Austin, Texas, Jan. 1988.

[AH88] J. Aspnes, and M. P. Herlihy, "Fast Randomized Consensus using Shared Memory," submitted to publication.

[ADS89] H. Attiya, D. Dolev, and N. Shavit, "A Bounded Probabilistic Shared-Memory Consensus Algorithm," unpublished manuscript.

[B88] S. Ben-David, "The Global Time Assumption and Semantics for Concurrent Systems," *Proc. 7th ACM Symp. on Principles of Distributed Computing*, 1988, pp. 223-231.

[Bl87] B. Bloom, "Constructing two-writer atomic registers," *Proc. 6th ACM Symp. on Principles of Distributed Computing*, 1987, pp. 249-259.

[BP87] J. E. Burns, and G. L. Peterson, "Constructing Multi-Reader Atomic Values from Non-Atomic Values," *Proc. 6th ACM Symp. on Principles of Distributed Computing*, 1987, pp. 222-231.

[CIL87] B. Chor, A. Israeli, and M. Li, "On Processor Coordination Using Asynchronous Hardware", *Proc. 6th ACM Symp. on Principles of Distributed Computing*, 1987, pp. 86-97.

[CMS85] B. Chor, M. Merritt and D. B. Shmoys, "Simple Constant-Time Consensus Protocols in Realistic Failure Models," *Proc. 4th ACM Symp. on Principles of Distributed Computing*, 1985, pp. 152-162.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus," *J. ACM 34*, 1987, pp. 77-97.

[DGS88] D. Dolev, E. Gafni, and N. Shavit, "Toward a Non-Atomic Era: L-Exclusion as a Test Case," *Proc. 20th Annual ACM Symp. on the Theory of Computing*, 1988.

[DS89] D. Dolev, and N. Shavit, "Bounded Concurrent Time-Stamp Systems Are Constructible!" *Proc. 21th Annual ACM Symp. on Theory of Computing*, 1989, to appear.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Processor," *J. ACM 32*, 1985, pp. 374-382.

[H88] M. P. Herlihy, "WaitFree Implementations of Concurrent Objects," *Proc. 7th ACM Symp. on Principles of Distributed Computing*, 1988, pp. 276-290.

[IL88] A. Israeli and M. Li, "Bounded Time Stamps," *Proc. 28th Annual IEEE Symp. on Foundations of Computer Science*, 1987, pp. 371-382.

[L86a] L. Lamport, "On Interprocess Communication. Part I: Basic Formalism," *Distributed Computing 1, 2* 1986, 77-85.

[L86b] L. Lamport, "On Interprocess Communication. Part II: Algorithms," *Distributed Computing 1, 2* 1986, pp. 86-101.

[L86c] L. Lamport, "The Mutual Exclusion Problem. Part I: A Theory of Interprocess Communication," *J. ACM 33, 2* 1986, pp. 313-326.

[L86d] L. Lamport, "The Mutual Exclusion Problem. Part II: Statement and Solutions," *J. ACM 33, 2* 1986, pp. 327-348.

[LV88] M. Li, and P. Vitanyi, "Uniform Construction for Wait-Free Variables," unpublished manuscript, 1988.

[LA87] M. G. Loui, and H. Abu-Amara, "Memory Requirements for Agreement Among Unreliable Asynchronous Processes", *Advances in Computing Research,* vol. 4, 1987, pp. 163-183.

[N87] R. Newman-Wolfe, "A Protocol for Wait-free Atomic, Multi Reader Shared Variables," *Proc. 6th ACM Symp. on Principles of Distributed Computing,* 1987, pp. 232-248.

[P83] G. L. Peterson, "Concurrent Reading While Writing," *ACM TOPLAS 5, 1* 1983, pp. 46-55.

[PB87] G. L. Peterson, and J. E. Burns, "Concurrent Reading While Writing II : The Multi-Writer Case," *Proc. 28th Annual IEEE Symp. on Foundations of Computer Science,* 1987, pp. 383-392.

[P89] S. Plotkin, "Sticky Bits and the Universality of Consensus," to appear in *Proc. 8th ACM Symp. on Principles of Distributed Computing.*

[S88] R. Schaffer, "On the Correctness of Atomic Multi-Writer Registers," MIT/LCS/TM-364, June 1988.

[SAG87] A. K. Singh, J. H. Anderson and M. G. Gouda, "The Elusive Atomic Register Revisited," *Proc. 6th ACM Symp. on Principles of Distributed Computing,* 1987, pp 206-221.

[VA86] P. Vitanyi, and B. Awerbuch, "Atomic Shared Register Access by Asynchronous Hardware," *Proc. 27th Annual IEEE Symp. on Foundations of Computer Science,* 1986, pp. 233-243.