Efficient Asynchronous Consensus with the Weak Adversary Scheduler

Yonatan Aumann*

Abstract

We consider the problem of asynchronous consensus with a weak dynamic adversary scheduler. We provide the first algorithm to obtain $\tilde{O}(n)$ total work in the weak adversary model using only *single-writer* registers. For the *multi-writer* setting we give an $O(\log n)$ work-per-processor algorithm, improving upon the previous $O(\log^2 n)$ bound. The adversary model considered is the *content oblivious adversary* model, which assumes that the adversary does not know the content of a register until it is read by some processor [13].

1 Introduction

Consensus is one of the most studied problems in distributed computing. In a landmark work, Fisher, Lynch and Peterson [16] proved the impossibility of deterministic asynchronous consensus, even with a single fault. Thus, randomization is introduced. Abrahamson [1] was the first to provide a randomized, exponential time, solution to the consensus problem assuming the standard strong adversary model. Subsequently, many polynomial-time algorithms were published [6, 19, 10, 4, 12, 8], reducing the complexity from $O(n^4)$ [6], to $\tilde{O}(n^2)$ [12, 8]. Still, no algorithm managed to cross the $\tilde{\Omega}(n^2)$ barrier. Recently, Aspnes proved that this is no accident. In [5] Aspnes proves that any protocol for asynchronous consensus with the strong dynamic adversary necessitates $\Omega(n^2/\log^2 n)$ work. This proves that the known results are near optimal. However, from a practical point of view the results are still unsatisfactory; $\tilde{\Omega}(n^2)$ work is unacceptable in many settings. In the words of Aspnes [5]:

> For those of us who like to work with the adaptive adversary, randomization has given only a temporary reprieve from the consequences of Fischer, lynch and Paterson's impossibility proof for deterministic consensus with faulty processors. Theorem 6 means that even though we can solve consensus using randomization, we cannot hope to solve it quickly without a small upper bound on the number of failures, built-in synchronization primitives, or restrictions on the power of the adversary.

Can anything be done? Aspnes continues:

> Fortunately, there are a number of natural restrictions on the adversary that allow fast consensus protocols without eliminating the faults that we might reasonably expect to observe in real systems. One plausible approach is to limit the knowledge the adversary has of register contents, to prevent it from discriminating against coin-flips it doesn't like. Various versions of this can be found in the consensus work of Chor, Israeli, and Li [14] and Abrahamson [1] and in the $O(n \log^2 n)$ total work protocol of Aumann and Bender [11] and the $O(\log^2 n)$ work-per-processor protocol of Chandra [13].

Thus, the weak adversary model seems to provide the desired solution. However, a close look at the related work reveals that even with the weak adversary model, no known algorithm manages to cross the $\tilde{\Omega}(n^2)$ barrier with this assumption alone; Abrahamson's protocol [1] requires $O(n^2 \log n)$ work, while the protocols of [11] and [13] do indeed achieve $\tilde{O}(n)$ work, but have the extra assumption of multi-writer registers. In this paper we provide the first protocol which, assuming the weak adversary model obtains consensus in $\tilde{O}(n)$ work using only *single-writer* registers. We obtain:

Theorem 1 *Assuming: an asynchronous shared memory system; single-writer / multi-reader, atomic read / write registers; a content oblivious adversary scheduler. Consensus can be obtained in expected $O(n \log^4 n)$ total number of shared memory accesses, and maximum expected $O(n)$ work per processor.*

*Department of Mathematics and Computer Science, Bar-Ilan University, Ramat-Gan, 92500, Israel, Email: aumann@cs.biu.ac.il

The *content oblivious scheduler* is the weak adversary scheduler as defined in [13]. Combined with Aspnes's lower bound, Theorem 1 shows that the weak adversary is strictly weaker than the strong one, even for single writer registers. We also show that both the *latency competitive ratio* and the *throughput competitive ratio* of our algorithm are $O(\log^4 n)$ (see [2, 9]).

The Model. We assume a shared memory asynchronous system with n processors, $P_1, \ldots, P_n$, and atomic read/write registers. The schedule of the system is described (posteriori) as a function $S : \mathcal{N} \rightarrow \{1, \ldots, n\}$, where $S(\pi) = j$ means that processor P_j performed the π-th step in the system. The schedule function is assumed to be determined by an *adversary scheduler*. The weak adversary scheduler comes in several incarnations, with some variation between them. In this work we follow the [13] definition, which we call the *content oblivious* adversary model. The model assumes a dynamic adversary scheduler which at each step decides on the next processor to operate. For this decision, the adversary is provided with the entire history of the dynamics of the system. However, it is assumed that the adversary has no knowledge of the actual *value* stored in a memory location as long as no processor read (fetched) this value. It is assumed that *writing* a bit to shared memory does not reveal its value. In particular, the adversary is not provided with the values of random bits written to shared memory until they are read by another processor. The rational behind these assumptions is that the asynchronous behavior of a system (e.g. system failures, page faults) is likely to be affected by *where* processors read and write, but not by *what* values are being written. Once a value is being *read* by a processor, it may be used in ways which can affect the dynamic behavior of the system. The algorithm code and the initial input values are assumed to be known to the adversary.

Efficiency is measured by the total amount of work performed in the system, summed up over all processors (coined *collective latency* in [7]). Following [19, 9] and others, we count only shared memory accesses (reads and writes). We also bound the expected work per-processor. We assume $O(n)$ size registers.

Collect Procedures. Consider the case where each processor has a private value written in its shared memory register. The *collect* primitive, first abstracted in [19], is a procedure by which one processor obtains the values of all other processors. Clearly, if several processors must perform a *collect* over the same set of registers, then they may cooperate in gathering the values. Thus, we have the notion of a *cooperative collect* procedure. Saks, Shavit, and Woll [19] showed that an efficient implementation of *cooperative collect*, together with a weak random coin, results in an efficient consensus protocol. In their result, [19] require that the *cooperative collect* procedure provide the following *freshness* guarantee: if the values in the registers may change over time, then the value obtained by processor P_i for P_j must be present in P_j's register at least once between the time P_i started its *collect* and the time it ended the *collect*. Unfortunately, if processors operate one after the other, any *fresh-collect* procedure necessitates n^2 work. Thus, any consensus algorithm based on *fresh-collect* takes $\Omega(n^2)$ work. Interestingly, we show that freshness is *not* necessary for consensus. We show that a much weaker *collective freshness* guarantee suffices. *Collective-freshness* means that the value obtained by P_i for P_j appeared in P_j's register at some time after *some* processor (not necessarily P_i) started the collect. Thus, we seek an efficient *collective-freshness collect* procedure. For this we build upon the elegant *collect* procedure of Aspnes and Hurwood [7], which obtains $O(n \log^3 n)$ work in the case where values never change. We show that an adaptation of the [7] protocol gives an $O(n \log^3 n)$ work algorithm for the *collective-freshness collect* problem, regardless of the schedule. Building upon this *collective-freshness collect* protocol (and some constructs drawn from [13]) we obtain an $O(n \log^4 n)$ total work protocol for the single-writer setting.

A Multi-Writer Algorithm

We also consider the multi-writer/multi-reader register setting, assuming the content oblivious adversary scheduler. For this setting we provide a protocol with $O(\log n)$ expected number of operations per-processor. We obtain:

Theorem 2 *Assuming: an asynchronous shared memory system; multi-writer / multi-reader, atomic read / write registers; a content oblivious adversary scheduler. Consensus can be obtained in expected $O(\log n)$ operations per processor.*

This improves upon the $O(\log^2 n)$ result of Chandra [13]. We note that in this case, following [13], we count all operations (not only shared memory accesses), and use $O(\log n)$ size registers.

1.1 Related Work.

Fischer, Lynch, and Paterson [16] prove the impossibility of deterministic asynchronous consensus in the message-passing model, even if only one processor fails. Chor, Israeli, and Li [14] and Loui and Abu-Amara [18] show that the same holds in the shared-memory model (also see [15]). A randomized solution for a weak model was first presented in [14]. Abrahamson [1] provides the first (exponential) solution for the strong adversary model, and a polynomial time algorithm for the weak model. A sequence of papers followed, providing improved bounds for the strong adversary model. Aspnes and Herlihy [6] introduce the notion of the *weak random coin* and give the first polynomial solution ($O(n^4)$) using unbounded registers. Attiya, Dolev, and Shavit [10] provide a polynomial, bounded-registers solution. Aspnes [3] gives an $O(n^2(p^2+n))$ bounded-register solution, where p is the number of active processors. Bracha and Rachman [12] give an $O(n^2 \log n)$, unbounded-registers solution. Finally, Aspnes and Waarts [8] provide a solution of $O(n \log^2 n)$ steps *per processor* using unbounded registers. All these algorithms use single writer registers.

The quest for more efficient algorithms using a weaker model was re-examined in [11] and [13]. Aumann and Bender [11] define the *value oblivious adversary* and provide an $O(n \log^2 n)$ total work protocol for this model, assuming multi-writer registers. Chandra [13] defines the adversary as described above, and provides an algorithm with expected $O(\log^2 n)$ operation per-processor, assuming multi-writer registers. We note that the [13] adversary model (and similarly that of [11]) is weaker than that of [14, 1], in the following sense. The [14, 1] model specifies that the adversary does not know the value of a random bit until it is written to shared memory. The [13] model provides that the random value is not known to the adversary, even when already in shared memory, until it is read by some processor. The model used in this paper is that of [13].

The *collet problem* was first abstracted and applied to consensus by Saks, Shavit and Woll [19]. A theory of competitive analysis for the distributed setting, together with competitive implementations of *collect* are discussed in [2] and [9]. Aspnes and Hurwood [7] provide solutions for various variants of the collect problem, assuming the weak adversary model. In particular, they give an $O(n \log^3 n)$ implementation of the *collect*, and an almost tight $\tilde{\Theta}(\sqrt{n})$ bound on the throughput competitiveness of *fresh-collect*.

2 The Single Writer Algorithm

2.1 The General Structure

The overall structure of the algorithm draws upon ideas from [13]. The protocol starts with each processor, P_i, writing its private input value, v_i, to shared memory. In addition, each processor P_i generates $\log n$ sequences of $\Theta(\log n)$ random bits each, and writes them to a two-dimensional array Random$[i][\cdot][\cdot]$ in shared memory.[1] Following this initialization phase, processors follow a recursive procedure to decide on a single *winner* processor. The consensus value is the input value of the winner. The details follow.

W.l.o.g. assume n is a power of 2. Suppose a set $S = \{P_{low}, \ldots, P_{high}\}$ of processors wishes to choose a winner. If S contains a single processor than this processor is the winner of the set. Otherwise, the set S is split into two equal size subsets S_0, S_1. Recursively, winners w_1 and w_1, are chosen for the subsets, S_0 and S_1, respectively. Then, using a *binary consensus protocol* (Section 2.2) the processors decide on the winner for the entire set. In the *binary consensus protocol* the processors of S_0 opt for w_0 and those of S_1 for w_1. A description of the protocol appears in Figure 1.

2.2 The Binary Consensus Protocol

The *binary consensus protocol* operates on a set of m processors $S = \{P_{low}, \ldots, P_{high}\}$, each with a private input value. (In the description of the protocol we disregard processors not in S.) The processors are partitioned into two sets S_0 and S_1. All processors of S_β, $\beta = 0, 1$, share the same input value, w_β. There are two arrays, Random$[w_\beta][level][\cdot]$, $\beta = 0, 1$, in shared memory, each consisting of $c \log n = \Theta(\log n)$ *fresh* random bits. The outcome of the protocol is a value $w \in \{w_0, w_1\}$, agreed upon by all processors completing the protocol.

The protocol for the binary consensus is based on the fundamental protocol of [6]. The presentation here follows that of [19]. The crucial difference is in the introduction of a new *collect* procedure. A full description of the protocol for the binary consensus is given in Figure 2. For each processor, P_i, there are two arrays, propose$[i][\cdot]$ and agree$[i][\cdot]$, to which it may write. In addition, there are m variables input$[i]$, $i = low, \ldots, high$, one for each processor. All variables and arrays are in shared memory and assumed to be initialized to NULL. The protocol uses a procedure COLLECT$(X[all])$, where $X[\cdot]$ is an array (propose$[\cdot][r]$ or agree$[\cdot][r]$, where r is an index, or input$[\cdot]$). The procedure COLLECT$(X[all])$ returns an array, $view(X[\cdot])$, of size m. Intuitively, the value of $view(X[j])$ corresponds to the value appearing in $X[j]$. The subroutine provides the following guarantee:

[1] Here and throughout this section, the first index of an array specifies the ID of the processor which may write the entry. Thus, Random$[i][\cdot][\cdot]$ is a two dimensional array writable by P_i; propose$[i][\cdot]$ is a one dimensional array writable by P_i; and input$[i]$ a single variable writable by P_i.

```
SINGLE-WRITE-CONSENSUS(v_i)   (for processor P_i)
for j := 1 to log n do
        for ℓ := 1 to c log n do
                WRITE(Random[i][j][ℓ] ← random-bit)
WRITE(input[i] ← v_i)
        { initializing complete }
win := CHOOSE-WINNER(1, n, 1)
return input[win]


CHOOSE-WINNER(low, high, level)   (for processor P_i)
if high = low then                                                        { bottom of recursion }
        return i
avg := (high + low)/2                                                     { else - recurse }
if i < avg then
        win := CHOOSE-WINNER(low, ⌊avg⌋, level + 1); β := 0 ;
else
        win := CHOOSE-WINNER(⌈avg⌉, high, level + 1) ; β := 1 ;
win := BINARY-CONSENSUS(win, β, level)   (with processors P_low, ..., P_high)      { compete }
return win
```

Figure 1: Single Writer Consensus Procedure

```
SINGLE-WRITER-BINARY-CONSENSUS(w_β, β, level)   (for processor P_i ∈ S_β)

1  WRITE(input[i] ← w_β)
2  myvalue := β
3  decide := False ; r := 0
4  repeat
5          r := r + 1
6          if r > c log n                          { no more random bits in Random[w_β][level][·] }
7                  then run alternate consensus protocol^a to agree on myvalue. Use current myvalue as entry value.
8          WRITE(propose[i][r] ← myvalue)
9          view(propose[·][r]) := COLLECT(propose[all][r])
10         if both 0 and 1 appear in view(propose[·][r])
11                 then WRITE(agree[i][r] ← 'no')
12                 else WRITE(agree[i][r] ← 'yes')
13         view(agree[·][r]) := COLLECT(agree[all][r])
14         if 'no' appears in view(agree[·][r]) then
           begin
15                 if exists k s.t. view(agree[k][r]) = 'yes'
16                         then myvalue := view(propose[k][r])
17                         else myvalue := Random[w_β][level][r]
           end
18         else decide := True
19 until decide
20 view(propose[·][1]) := COLLECT(propose[all][1])^b
21 find k s.t. view(propose[k][1]) = myvalue
22 return input[k]
```

[a] Any polynomial time algorithm (e.g. [8]).

[b] Here the COLLECT() is performed again from scratch. Results from the COLLECT(propose[all][1]) of line 9 are not used.

Figure 2: Single Writer Binary Consensus Procedure

Collective freshness: Let t_0 be the first time COLLECT($X[all]$) was ever evoked by *any* processor. For any j and i, the value $view(X[j])$ obtained by P_i in response to COLLECT($X[all]$) appeared in $X[j]$ at some time after t_0 and before P_i completed its COLLECT($X[all]$).

In particular, if P_j wrote the value v to $X[j]$ before *any* processor evoked COLLECT($X[all]$) and did not subsequently overwrite it, then $view(X[j]) = v$ for all processors evoking COLLECT($X[all]$). Note that *collective freshness* is much weaker than the freshness property required in [19].

Implementing the COLLECT() procedure. The COLLECT() procedure is implemented using a variant of the *collect* algorithm of [7]. In the [7] *collect* protocol there are m processors, $P_\ell, \ldots, P_k$, each with a private value, x_i, stored in the shared memory section writable by P_i. In the [7] setting, the values are assumed *not* to change. The objective is to have each processor know the entire array of values $(x_i)_{i=\ell}^{k}$. The [7] algorithm completes the collect in expected $O(m \log^3 m)$ reads and writes, regardless of the asynchronous behavior of the system (assuming any processor which knows all values exits the procedure). We transform this *collect* algorithm to a *collective-freshness collect* algorithm. The main difference in the case of *collective-freshness* is that the input values, x_i, may change during the *collect* (in our setting, they may change from NULL to v_i). However, in this case *either* value is fine. Accordingly, for the *collective-freshness collect* procedure we use the [7] *collect* algorithm, with the additional rule that if a processor obtains two different values for the same processor then it may choose to retain either value. A processor completes the procedure once it gets *some* value for each of the processors. Thus, *collective-freshness collect* is completed in $O(m \log^3 m)$ memory accesses.

We also bound the maximum number of operations performed by any single processor, as follows. Consider the naive *collect* protocol where each processor reads each of the other processors' registers, one-by-one. This naive protocol takes $m - 1$ work per processor. Thus, to bound the maximum work per processor, we have each processor alternate between taking a one step of the naive protocol and one step of the more complex protocol described above. This way we guarantee both $O(m \log^3 m)$ total work summed up over all processors, and $O(m)$ work for any single processor.

Guarding against long executions. We shall prove that for any level, with high probability, the $\Theta(\log n)$ random bits of Random$[w_\beta][level][\cdot]$, $\beta = 0, 1$ suffice. However, in are rare event, all bits may be used before consensus is obtained. In this rare event we run an alternate consensus protocol, using any standard polynomial consensus algorithm (lines 6-7). This addition does not change the expected complexity of the algorithm. When a processor enters the alternate protocol, it uses its *current myvalue* as its input value to the protocol. We shall prove that this guarantees consensus. In particular, we prove that if any processor reaches a decision on value b without using the alternate protocol, then all processors participating in the alternate protocol enter it with value b (see [17] for a similar construction for Byzantine Agreement).

2.3 Analysis

We now prove that COLLECT() with the *collective-freshness* guarantee suffices to insure correctness of the binary consensus procedure, and that the procedure completes in expected of $O(1)$ rounds, i.e. $O(m \log^3 m)$ total work.

Claim 1 *1. If all processors that start round r enter with the same value b for myvalue, then all processors decide in the round, and exit the loop with $myvalue = b$.*

2. *All processors P_i that write 'yes' to* agree$[i][r]$ *wrote the same value to* propose$[i][r]$.

3. *All processors that see at least one 'yes' in their* $view$(agree$[\cdot][r]$) *end the round with the same myvalue.*

4. *If any processor has decide =* True *in round r with $myvalue = b$ then all processors end the round with $myvalue = b$.*

5. *Suppose that round r is the first round in which any processor obtains decide =* True*, and that this processor reaches line 20 with $myvalue = b$, then any other processor decides in round r', $r' \leq r + 1$, and also reaches line 20 with $myvalue = b$.*

6. *Any processor that reaches line 21 with $myvalue = b$, finds at least one entry in* $view$(propose$[\cdot][1]$) *with the value b.*

7. *All processors complete the protocol with the same return value.*

Proof:

1. For all processors, only b (and possibly NULL) will appear in $view$(propose$[\cdot][r]$). Thus, they all write agree$[i][r]$ = *'yes'* in line 12. Thus, they all set *decide* := True in line 18. (If r is the round of the

alternate protocol (line 7) then the claim follows from the definition of consensus).

2. Let P_{i_0} be the first processor to ever evoke COLLECT(propose[*all*][r]), and let b be the value P_{i_0} wrote to propose[i_0][r]. Note that this write occured before *any* processor evoked COLLECT(propose[*all*][r]). Thus, *view*(propose[i_0][r]) $= b$ for all processors. Thus, any processor that did not write b to propose[i][r] will see both 0 and 1 in *view*(propose[$\cdot$][r]).

3. By (2) and lines 15-16.

4. Let P_{i_1} be the first processor to ever evoke COLLECT(agree[*all*][r]), and let v be the value written by P_{i_1} to agree[i_1][r]. This write occured before any processor evoked COLLECT(agree[*all*][r]). Thus, v appears in the *view*(agree[$\cdot$][r]) for all processors. Consider two cases. If $v =$ *'no'* then no processor will decide in this round, and there is nothing to prove. If $v =$ *'yes'* then by (3) all processors end the round with *myvalue* $= b$. (If r is the round of the alternate protocol (line 7) then the claim follows from the definition of consensus).

5. By (4) all processors deciding in round r reach line 20 with *myvalue* $= b$. Consider those advancing to round $r+1$. By (4), they all enter with *myvalue* $= b$. Thus, by (1) they all decide in round $r+1$ and exit with *myvalue* $= b$.

6. Let P_{i_2} be the first processor to ever evoke COLLECT(propose[*all*][r]) in line 20, and suppose that *myvalue* $= b$ for P_{i_2} at line 20. Suppose that at the time P_{i_2} reaches line 20 propose[i][1] $\neq b$ for all i. Then, by (1), P_{i_2} would exit after the first round with *myvalue* $= 1-b$, in contradiction. Thus, by the time the scan in line 20 is evoked for the first time, there is at least one i with propose[i][1] $= b$. This entry is obtained by all scans of line 20.

7. By 5 all processors that reach line 20 have the same *myvalue* $= b$. By definition, all processors proposing b in round 1 have the same input[i] $= w_\beta$ value.

∎

Next, we bound the expected number of rounds.

Claim 2 *For any round r with probability $\geq 1/4$ all processors complete the round with the same myvalue, regardless of the entry values.*

Proof: Let P_{i_0} be the first processor to ever evoke COLLECT(propose[*all*][r]), and let $b =$ propose[i_0][r]. Then, all processors see *view*(propose[i_0][r]) $= b$. Thus, any processor completes the round with *myvalue* $\in$ $\{b$, Random[w_0][*level*][r], Random[w_1][*level*][r]$\}$. This is because all processors that write *'yes'* to agree[i][r] must have propose[i][r] $= b$ (Claim 1-(2)). Hence, for any processor, either decides on b in the round, or it sets *myvalue* $= b$ in line 16, or sets *myvalue* $=$ Random[w_β][*level*][r] in line 17. Note that P_{i_0}, and hence b, are determined before *any* processor reads either Random[w_0][*level*][r] or Random[w_1][*level*][r]. Thus, with probability at least $1/4$, $b =$ Random[w_0][*level*][r] $=$ Random[w_1][*level*][r], in which case all processors end the round with the same *myvalue*. ∎

We thus obtain:

Theorem 3 *The binary-consensus algorithm guarantees consensus. The protocol completes in expected $O(1)$ rounds, $O(m \log^3 m)$ total number of shared memory accesses, and at most expected $O(m)$ memory accesses per processor.*

Which implies Theorem 1 (restated):

Theorem 1 *The protocol of Figure 1 guarantees consensus. The expected total number of shared memory accesses is $O(n \log^4 n)$, and at most $O(n)$ per processor.*

Proof: At level i of the recursion, there are 2^i binary consensus algorithms performed, each with expected $O(m \log^3 m)$ work, with $m = n/2^i$. There are $\log n$ levels. Thus, the total expected work is $O(n \log^4 n)$. Any given processor performs expected $O(n/2^i)$ work at level i. Thus, the total work per processor is $O(n)$. ∎

Throughput competitiveness was defined in [9].

Theorem 4 *The throughput competitive ratio of the consensus algorithm is $O(\log^4 n)$.*

Proof: Given a schedule S, and time interval I, we count the number of consensus procedures our algorithm completes versus the number a *champion* distributed algorithm can complete. We assume that a new consensus procedure is not started until at least one processor finished the previous one. (However, some processors may still be working also on "old" consensus procedures. In this case we allow these slow processors to work on multiple consensuses concurrently.) Under this assumption, the champion algorithm must perform at least $O(n)$ memory reads before it can start a new consensus procedure. This is because otherwise there is at least one processor which is not accessed between the two procedures, in which case its state cannot be known, and no processor can decide on a value. On the other hand, our algorithm completes a consensus procedure every $O(n \log^4 n)$ memory accesses. Thus, the throughput ratio is $O(\log^4 n)$. ∎

Similarly we have:

Theorem 5 *The latency competitive ratio ([2]) of the consensus algorithm is* $O(\log^4 n)$.

3 The Multi-Writer Algorithm

In this section we describe the protocol for the multi-reader/multi-writer register setting. We give an algorithm with expected $O(\log n)$ operations per processor. It should be noted that the complexity measure used in the section is slightly different from the one used in the previous section. Here we count all operations, not only shared memory accesses. This difference follows that of [19] and [13]. The algorithm is similar in flavor to that of [13]. The improvement is obtained by a more economical usage of the random bits.

The general structure of the protocol is similar to that of single-writer protocol. The main differences are in the implementation of arrays propose[·][·], agree[·][·], and input[·], and in a more economical use of the random bits.

In the implementation of the binary-consensus protocol, we use only *two* "propose" arrays: propose[0][·] and propose[1][·]. In any round r, a processor wishing to propose b does so by writing a "checkmark", $\surd$, to propose[b][r] ($b = 0, 1$). Similarly we have only two arrays agree[*'yes'*][·] and agree[*'no'*][·], and two input variables input[0] and input[1]. The COLLECT() operation is replaced by reading both relevant cells. All variables and arrays are initialized to NULL.

The random bits are maintained as follows. In the initializing phase each processor P_i generates *two* sequences of $\Theta(\log n)$ random bits each, Random[i][·] and L-Random[i][·]. Now consider the recursive procedure. Once the recursion has unfolded, we may view the structure of the computation as a tree, which we call the *recursion tree.* The essential observation is that, with high probability, along any *path* in the recursion tree there are a total of $O(\log n)$ consensus *rounds*. Furthermore, for any processor P_i, the nodes of the tree where P_i was competing for *winner* constitute a path. Thus, for each processor P_i we use the $\Theta(\log n)$ random bits of Random[i][·] for all nodes where P_i competes for winner. The only problem is that processors must agree on the current location in Random[i][·]. To this end, at each node of the recursion tree, following the agreement on the winner, we run an additional POSITION-CONSENSUS protocol, by which processors decide on the current *position* in Random[*win*][·] (where *win* is the winner of at the node). For this POSITION-CONSENSUS protocol we use the random bits of L-Random[*loser*][·], where *loser* is the ID of the processor which *lost* the competition at the node. Since any processor can be the loser at most once, these bits are fresh.

A full description of the protocol appears in Figures 3, 4 and 5.

In the POSITION-CONSENSUS protocol there are $c \log n$ arrays propose[p][·], $p = 1, \ldots, c \log n$, each of size $c \log n$. All arrays are initialized to NULL. A processor proposes *position* $= p$ in round r by placing a checkmark in propose[p][r]. A key point in the algorithm is that there are at most two possible proposals for *position.* The variable *high* denotes the higher of the two proposals.

3.1 Analysis

First we consider the BINARY-CONSENSUS protocol. The following claim is proved similar to Claim 1.

Claim 3
1. *If all processors that start round r enter with the same value for myvalue, then all processors decide in the round, and exit the loop with the same myvalue.*
2. *All processors that write to* agree[*'yes'*][r] *started the round with the same myvalue.*
3. *All processors that write to* agree[*'yes'*][r] *write the same value.*
4. *All processors that see* agree[*'yes'*][r] $\neq$ NULL *complete the round with the same myvalue.*
5. *If any processor has decide* $=$ True *in round r with myvalue $= b$ then all processors end the round with myvalue $= b$.*
6. *Suppose r is the first round that any processor obtains decide* $=$ True, *and that this processor reaches line 19 with myvalue $= b$, then any other processor decides in round r', $r' \leq r + 1$, and also reaches line 19 with myvalue $= b$.*

The following claim is analogous to Claim 2.

Claim 4 *For any round r with probability $\geq 1/4$ all processors complete the round with the same myvalue, regardless of the entry values.*

Thus, BINARY-CONSENSUS is correct and completes in expected $O(1)$ rounds. Next, we analyze the POSITION-CONSENSUS protocol. The correctness of the protocol follows from the following sequence of claims.

```
Multi-Writer-Consensus(v_i)   (for processor P_i)
for ℓ := 1 to c log n do
        Write(Random[i][ℓ] ← random-bit)
        Write(L-Random[i][ℓ] ← random-bit)
Write(input[i] ← v_i)
        { initializing complete }
< win, position >:= Choose-Winner(1, n, 1)
return input[win]


Choose-Winner(high, low, position)  (for processor P_i)
if high = low then                                                    { bottom of recursion }
        return i
avg := (high + low)/2                                                 { else - recurse }
if i < avg then
        < win, position >:= Choose-Winner(low, ⌊avg⌋, position); β = 0 ;
else
        < win, position >:= Choose-Winner(⌈avg⌉, high, position); β = 1 ;
< win, position >:= binary-consensus(win, β, position) (with processors P_low, ..., P_high)   { compete }
return < win, position >
```

Figure 3: Multi Writer Consensus Procedure

```
Multi-Writer-Binary-Consensus(w_β, β, position)

Write(input[β] ← w_β)
myvalue := β
decide := False ; r := position
repeat
        r := r + 1
        if r > c log n                                                { no more random bits }
                then run alternate consensus protocol^a to agree on myvalue. Use current myvalue as entry value.
        Write(propose[myvalue][r] ← √)
        Read propose[1 − myvalue][r]
        if propose[1 − myvalue][r] = √                                { both 0 and 1 are proposed }
                then Write(agree['no'][r] ← √)
                else Write(agree['yes'][r] ← myvalue)
        if agree['no'][r] = √ then
        begin
                if agree['yes'][r] ≠ NULL
                        then myvalue := agree['yes'][r]
                        else myvalue := Random[w_β][r]
        end
        else decide := True
until decide
position := Position-Consensus(input[1 − myvalue],r)
return < input[myvalue], position >

^a Any polynomial time algorithm (e.g. [8]).
```

Figure 4: Multi Writer Binary Consensus Procedure

```
POSITION-CONSENSUS(LoserID,position)

1  if LoserID ≠ NULL then WRITE(loser ← LoserID)
2  decide := False
3  r := 0
4  repeat
5        r := r + 1
6        if r > c log n                                                    { no more random bits }
7              then run alternate consensus protocol^a to agree on position. Use current position as entry value.
8        WRITE(propose[position][r] ← √)
9        Read propose[position − 1][r] and propose[position + 1][r]
10       if propose[position − 1][r] = √
11             then WRITE(agree['no'][r] ← √); high := position
12       else if propose[position + 1][r] = √
13             then WRITE(agree['no'][r] ← √); high := position + 1
14       else WRITE(agree['yes'][r] ← position)
15       if agree['no'][r] ≠ NULL then
         begin
16             if agree['yes'][r] = √ then position := agree['yes'][r]
17             else if L-Random[loser][r] = 0 then position := high
18                   else position := high − 1
         end
19       else decide := True
20 until decide
21 return position
```

[a] Any polynomial time algorithm (e.g. [8]).

Figure 5: Position Consensus Procedure

Claim 5 1. *At any level of the recursion, with winner win, if a processor starts the* POSITION-CONSENSUS *protocol with position* $= k$ *then all the bits* Random$[win][r]$, $r > k$, *where never read before.*

2. *At any level of the recursion, for any processor, the value of LoserID is either* NULL *or the ID of the loser at this level.*

3. *At any level of the recursion, any processor which completed the* BINARY-CONSENSUS *in more than one round enters the* POSITION-CONSENSUS *protocol with LoserID* $\neq$ NULL.

4. *For any round r if any processor finds* agree$['no'][r] = \surd$ *then loser* $\neq$ NULL *at all times after this read (this insures line 17 is well defined).*

5. *For any round r, if all processors start the round with the same value for position, then all processors decide in the round and exit with the same position value.*

6. *All processors that write to* agree$['yes'][r]$ *write the same value.*

7. *All processors that see* agree$['yes'][r] \neq$ NULL *end the round with the same value for position.*

8. *If any processor has decide* $=$ True *in round r with position* $= b$ *then all processors complete the round with position* $= b$.

9. *For any round r, all processors that write* agree$['no'][r]$ *have the same value for high.*

10. *For any round r with probability $1/2$ all processors complete the round with the same position value, regardless of the entry values.*

11. *The expected number of rounds is 2 and with high probability there are at most $O(\log n)$ rounds.*

Which implies Theorem 2 (restated):

Theorem 2 *For any processor the expected number of operations to complete the multi-writer consensus procedure is $O(\log n)$.*

Proof: Consider a leaf-to-root path in the recursion tree. At each level, the expected number of rounds is constant. Thus the total expected number of rounds is $O(\log n)$. As long as $r \leq c \log n$, each round takes a constant number of operations per processor. With probability $\geq 1 - n^{-4}$ the total number of rounds along the path does not exceed $20 \log n$. In this case, the random bits in Random[·][·] and L-Random[·][·] suffice. Otherwise, the alternate $O(n \log^2 n)$ protocol of [8]) is applied at most $\log n$ times, for a total of $O(n \log^3 n)$ operation per processor. Thus, for $c \geq 20$, the expected number of operations per processor is $O(\log n)$. ■

References

[1] K. Abrahamson. On achieving consensus using shared memory. In *Proceedings of the 7th Annual ACM Symposium on the Principles of Distributed Computing*, pages 291–302, 1988.

[2] M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts. A theory of competitive analysis for distributed algorithms. In *Proceedings of the 25th IEEE Symposium on Foundations of Computer Science*, pages 401–411, 1994.

[3] J. Aspnes. Time- and space-efficient randomized consensus. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pages 325–331, 1990.

[4] J. Aspnes. Time- and space-efficient randomized consensus. *Journal of Algorithms*, 14:414–431, 1993.

[5] J. Aspnes. Lower bounds for distributed coin-flipping and randomized consensus. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, 1997.

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

[7] J. Aspnes and W. Hurwood. Spreading rumors rapidly despite an advesary. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 143–151, 1996.

[8] J. Aspnes and O. Waarts. Randomized consensus in expected $O(n \log^2 n)$ operations per processor. In *Proceedings of the 33rd Annual Symposium on the Foundations of Computer Science*, pages 137–146, 1992.

[9] J. Aspnes and O. Waarts. Madular competitiveness for distributed algorithms. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pages 237–246, 1996.

[10] H. Attiya, D. Dolev, and N. Shavit. Bounded polynomial randomized consensus. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*, pages 281–294, 1989.

[11] Y. Aumann and M. Bender. Efficient asynchronous consensus with the value-oblivious adversary scheduler. In *Proceedings of the 23rd International Colloquium on Automata, Languages, and Programming*, 1996.

[12] G. Bracha and O. Rachman. Randomized consensus in expected $O(n^2 \log n)$ operations. In *Proceedings of the 5th International Workshop on Distributed algorithms.* Springer-Verlag, 1991.

[13] T.D. Chandra. Polylog randomized wait-free consenus. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, 1996.

[14] B. Chor, A. Israeli, and L. Ming. On processor coordination using asynchronous hardware. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 86–97, 1987.

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

[16] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed commit with one faulty process. *Journal of ACM*, 32(2):374–382, April 1985.

[17] O. Goldreich and E. Petrank. The best of both worlds: Guaranteeing termination in fast randomized byzantine agreement protocols. *Information Processing Letters*, 36:45–49, October 1990.

[18] H. Loui, M. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[19] M. Saks, N. Shavit, and H. Woll. Optimal time randomized consensus - making resilient algorithms fast in practice. In *Proceedings of the 2nd ACM-SIAM Annual Symposium on Discrete Algorithms*, pages 351–362, 1991.