Chapter 40

Optimal Time Randomized Consensus - Making Resilient Algorithms Fast in Practice*

Michael Saks† Nir Shavit‡ Heather Woll†

Abstract

In practice, the design of distributed systems is often geared towards optimizing the time complexity of algorithms in "normal" executions, i.e. ones in which at most a small number of failures occur, while at the same time building in safety provisions to protect against many failures. In this paper we present an optimally fast and highly resilient shared-memory randomized consensus algorithm that runs in only $O(\log n)$ expected time if $\sqrt{n}$ or less failures occur, and takes at most $O(\frac{n^3}{n-f})$ expected time for any f. Every previously known resilient algorithm required polynomial expected time even if no faults occurred. Using the novel consensus algorithm, we show a method for speeding-up resilient algorithms: for any decision problem on n processors, given a highly resilient algorithm as a black box, it modularly generates an algorithm with the same strong properties, that runs in only $O(\log n)$ expected time in executions where no failures occur.

*This work was supported by NSF contract CCR-8911388

†Department of Computer Science and Engineering, Mail Code C-014, University of California, San Diego, La Jolla, CA 92093-0114.

‡IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120.

1 Introduction

1.1 Motivation

This paper addresses the issue of designing highly resilient algorithms that perform optimally when only a small number of failures occur. These algorithms can be viewed as bridging the gap between the theoretical goal of having an algorithm with good running time even when the system exhibits extremely pathological behavior, and the practical goal (cf. [19]) of having an algorithm that runs optimally on "normal executions," namely, ones in which no failures or only a small number of failures occur. There has recently been a growing interest in devising algorithms that can be *proven* to have such properties [7, 11, 13, 22, 16]. It was introduced in the context of asynchronous shared memory algorithms by Attiya, Lynch and Shavit [7]. [1]

The *consensus* problem for *asynchronous shared memory systems* (defined below) provides a paradigmatic illustration of the problem: for reliable systems there is a trivial algorithm that runs in constant time, but there is provably no deterministic algorithm that is

[1] [11, 13, 22, 16] treat it in the context of *synchronous* message passing systems.

guaranteed to solve the problem if even one processor might fail. Using randomization, algorithms have been developed that guarantee an expected execution time that is polynomial in the number of processors, even if arbitrarily many processors fail. However, these algorithms pay a stiff price for this guarantee: even when the system is fully reliable and synchronous they require time at least quadratic in the number of processors.

1.2 The consensus problem

In the *fault-tolerant consensus problem* each processor i gets as input a boolean value x_i and returns as output a boolean value d_i (called its *decision value*) subject to the following constraints: Validity: If all processors have the same initial value, then all decision values returned are equal to that value; Consistency: All decision values returned are the same; and Termination: For each non-faulty process, the expected number of steps taken by the processor before it returns a decision value is finite.

We consider the consensus problem in the standard model of *asynchronous shared-memory systems*. Such systems consist of n processors that communicate with each other via a set of shared-registers. Each shared register can be written by only one processor, its owner, but all processors can read it. The processors operate in an asynchronous manner, possibly at very different speeds. In addition it is possible for one or more of the processors to halt before completing the task, causing a *fail-stop* fault. Note that in such a model it is impossible for other processors to distinguish between processors that have failed and those that are delayed but non-faulty. We use the standard notion of asynchronous time (see, e.g., [3, 17, 18, 20, 21]) in which one *time unit* is defined to be a minimal interval in the execution of the algorithm during which each non-faulty processor executes at least one step. Thus if during some interval, one processor performs 10 operations while another performs 100, then the elapsed time is at most 10 time units. Note that an algorithm that runs in time T under this measure of time, is guaranteed to run in real time $T \cdot \Delta$ where Δ is the maximum time required for a non-faulty processor to take a step.

Remarkably, it has been shown that in this model there can be no deterministic solution to the problem. This result was directly proved by [2, 9, 20] and implicitly can be deduced from [12, 15]. Herlihy [17] presents a comprehensive study of this fundamental problem, and of its implications on the construction of many synchronization primitives. (See also [23, 8]).

While it is impossible to solve the consensus problem by a deterministic algorithm, several researchers have shown that, under the assumption that each processor has access to a fair coin, there are randomized solutions to the problem that guarantee a probabilistic version of termination. Chor, Israeli, and Li [9] and Abrahamson [1] provided the first solutions to the problem, but in the first case the solution requires a strong assumption about the operation of the random coins available to each processor, and in the latter the expected running time was exponential in n. A breakthrough by Aspnes and Herlihy [4] yielded an algorithm that runs in expected time $O(\frac{n^3}{(n-f)})$ (here f is the (unknown) number of faulty processor); later Attiya, Dolev and Shavit [6] and Aspnes [5] acheived a similar running time with algorithms that use only bounded size memory.

1.3 Our results

In this paper, we present a new randomized consensus algorithm that matches the $O(\frac{n^3}{n-f})$ expected time performance of the above algorithms for $\sqrt{n} \leq f < n$, yet exhibits *optimal*[2] expected time $O(\log n)$ in the presence of $\sqrt{n}$ faults or less.

The starting point for our algorithm is a simplified and streamlined version of the Aspnes-Herlihy algorithm. From there, we reduce the

[2] A straightforward modification of the deterministic lower bound of [7] implies an $\Omega(\log n)$ lower bound.

running time using several new techniques that are potentially applicable to other share memory problems. The first is a method that allows processors to collectively scan their shared memory in expected time $O(\log n)$ time despite asynchrony and even if a large fraction of the processors are faulty. The second is the construction of an efficient *shared-coin* that provides a value to each processor and has the property that for each $b \in \{0,1\}$ there is a non-trivial probability that all of the processors receive value b. This primitive has been studied in many models of distributed computing (e.g., [24],[10]); polynomial time implementations of shared-coins for shared memory systems were given by [4] and [5]. By combining three distinct shared coin implementations using the algorithm interleaving method of [7], we construct a shared coin which runs in expected time $O(\log n)$ for executions where $f \leq \sqrt{n}$, and in $O(\frac{n^3}{n-f})$ expected time for any f.

The above algorithm relies on two standard assumptions about the characteristics of the system: (i) the atomic registers addressable in one step have size polynomial in the number of processors and (ii) the time for operations other than writes and reads of shared memory is negligible. We provide a variation of our consensus algorithm that eliminates the need for the above assumptions: it uses registers of logarithmic size and has low local computation time. This algorithm is obtained from the first one by replacing the randomized procedure for performing the collective scan of memory by a deterministic algorithm which uses a binary tree to collect information about what the processors have written to the vector.

In summary, our two different implementations of the global scan primitive give rise to two different wait-free solutions to the consensus algorithm. In the case that the number of failing processors, f, is bounded by $\sqrt{n}$, our first algorithm acheives the optimal expected time of $O(\log n)$ and our second algorithm acheives expected time $O(\log n + f)$, and in general, when f is not specially bounded, both algorithms run in expected time $O(\frac{n^3}{n-f})$.

Finally, using the fast consensus algorithm and the alternated-interleaving method of [7], we are able to prove the following powerful theorem: for any decision problem P, given any wait-free or expected wait-free solution algorithm $A(P)$ as a black box, one can modularly generate an expected wait-free algorithm with the same worst-case time complexity, that runs in only $O(\log n)$ expected time in failure-free executions.

The rest of the paper is organized as follows. In the next section we present a preliminary "slow" consensus algorithm which is based on the structure of the Aspnes - Herlihy algorithm. We show that this algorithm can be defined in terms of the two primitives, scan and shared-flip. In Section 4, we present a fast implementation of the scan primitive and in Section 5, we describe our fast implementations of the coin primitive. The last section describes the above-mentioned application and concludes with some remarks concerning extensions and improvements of our work. When possible, we give an informal indication of the correctness and timing analysis of the algorithms. Proofs of correctness and the timing analysis will appear in the final paper.

2 An Outline of a Consensus Algorithm

This section contains our main algorithm for the consensus problem, which we express using two simple abstractions:a *shared-coin*, which was used in [4] and a shared *write-once-vector*. Each has a "natural" implementation, which when used in the algorithm yields a correct but very slow consensus algorithm. The main contributions of this paper, presented in the two sections following this one, are new highly efficient randomized implementations for these primitives. Using these implementations in the consensus algorithm yields a consensus algorithm with the properties claimed in the introduction.

A *write-once vector* v consists of a set of n

memory locations, one controlled by each process. The location controlled by processor i is denoted v_i. All locations are initialized to a null value $\perp$ and each processor can perform a single *write* operation on the vector to the location it controls. Each processor can also perform one or more *scan* operations on the vector. This operation returns a "view" of the vector, that is, a vector whose i^{th} entry contains either the value written to v_i by processor i or is $\perp$. The key property of this view is that any value written to v_i before the scan began must appear in the view. A trivial $O(n)$-time implementation of a scan that ensures this property is: read each register of the vector in some arbitrary order and return the values of each read. Indeed, it would appear that any implementation of a *scan* would have to do something like this; as we will see later, if each processor in some set needs to perform a scan, then they can combine their efforts and achieve a considerably faster scan despite the asynchrony.

A *shared-coin* with *agreement parameter* δ is an object which can be accessed by each processor only through a call to the function *flip* applied to that object. Each processor can call this function at most once. The function returns a (possibly different value in {0,1} to each processor, subject to the following condition: For each value $b \in \{0,1\}$, the probability that all processors that call the function get the value b is at least δ, (and further this holds even when the probability is conditioned on the outcome of shared flips for other shared-coin objects and upon the events that happen prior to the first call to shared-flip for that object.) The simplest implementation of a shared-coin is just to have *flip* return to each processor the result of a local coin flip by the processor; this implementation has agreement parameter 2^{-n}.

The structure of our basic algorithm, presented in Figure 1, is a streamlined version of the algorithm proposed by [4] based on the two phase locking [1]. The algorithm proceeds in a sequence of rounds. In every round, each of the processors proposes a possible decision value, and all processors attempt to converge to the same value. (The reader should keep in mind that, due to the asynchrony of the system, processors are not necessarily in the same round at the same time.)

```
function consensus(my_value : input);
  begin
1:  decide := false; r := 1;
    repeat
2:   r := r + 1;
3:   write(proposed[r], my_value);
4:   prop_view := scan(proposed[r]);
5:   if both 0 and 1 appear in prop_view
6:    then write(check[r], 'disagree');
7:    else write(check[r], 'agree');
    fi;
8:   check_view := scan(check[r]);
9:   if 'disagree' appears in check_view then
      begin
10:    coin := shared_coin_flip(r);
11:    if for some p check_view[p]='agree'
12:     then my_value := prop_view[p]
13:     else my_value := coin;
      fi
      end
14:   else decide := true
    fi;
    until decide;
15: return my_value;
  end;
```

Figure 1: Main Algorithm — Code for P_i.

In each round r, a processor P_i publicly announces its proposed decision value by writing it to its location in a shared-array $proposed[r]$ (Line 3). It then (Line 4) performs a scan of the values proposed by the other processors, recording "agree" or "disagree" in its location of a second shared array $check[r]$ (Lines 6-7) depending on whether all of the proposed values it saw were the same. Next each processor performs a scan of $check[r]$ and acts as follows: 1) If it only sees "agree" values in $check[r]$ then it completes the round and decides on its own value; 2) If it sees at least one "agree" and at least one "disagree" it adopts the value written to $proposed[r]$ by one of the processors that wrote "agree"; 3) If it sees only "disagree" then

it changes its value to be the result of a coin-flip.

The formal proof that the algorithm satisfies the validity and consistency properties (contained in the full version of the paper) is accomplished by establishing a series of properties listed in the lemma below.

Lemma 2.1 *For each round r:*

1. *If all processors that start round r have the same myvalue then all non-faulty processors will decide on that value in that round.*
2. *All processors that write agree to check[r] wrote the same value to proposed[r].*
3. *If any processor decides on a value v in round r, then each non-faulty processors that completes round r sees at least one sf agree in its scan of check[r].*
4. *Every processor that sees at least one* agree *value in its scan of check[r] and completes round r with the same myvalue.*
5. *If any processor decides on a value v in round r then each non-faulty processors will decide on v during some round $r' \leq r+1$.*

Theorem 2.2 *The algorithm of figure 1 satisfies both the validity and consistency properties of the consensus problem.*

To prove that the algorithm also satisfies the (almost sure) termination property, define E_r to be the event that all processors that start round E_r have the same *myvalue*. From Lemma 2.1 it follows that if E_r holds then each non-faulty processor decides no later than round $r+1$. Using the lemma and the property of the shared-coin, it can be shown that for any given round greater than 1, the conditional probability that E_r holds given the events of all rounds prior to round $r-1$ is at least δ (where δ is the parameter of the shared-coin). This can be used to prove the following:

Lemma 2.3 *1. With probability 1,there exists an r such that E_r holds.*

2. The expected number of rounds until the last non-faulty processor decides is at most $1+1/\delta$.

Furthermore it can be shown that the expected running time of the algorithm can be estimated by $1+1/\delta$ times the expected time required for all non-faulty processors to complete a round.

For the naive implementations of the shared-coin and scan primitives, this yields an expected running time of $O(n2^n)$.

3 Interleaving Algorithms

The construction of our algorithms is based on using a variant of the *alternated-interleaving* method of [7], a technique for integrating wait-free (resilient but slow) and non-wait-free (fast but not resilient) algorithms to obtain new algorithms that are both resilient and fast.

The procedure(s) to be alternated are encapsuled in **begin-alternate** and **end-alternate** brackets or in **begin-alternate** and **end-alternate-and-halt** brackets (see Figure 3). The implied semantics are as follows. Each process P_i is assumed at any point in its execution to maintain a current list of procedures to be alternately-executed. Instead of simply executing the code of the listed procedures one after another, the algorithm alternates strictly between executing single steps from each. The **begin end-alternate** brackets indicate a new set of procedures (possibly only one) to be added to the list of those currently being alternately executed. The procedure or program from which the alternation construct is called continues to execute once the alternated procedures are added to the list, and can terminate even if the alternated procedures have not terminated.

For any subset of procedures added to the list in the same **begin end-alternate** state-

ment, *all* are deleted from the list (their execution is stopped) upon completion of any one of them. This however does not include any "sibling procedures," i.e. those spawned by **begin end-alternate** statements inside the alternated procedures themselves. Such sibling procedures are not deleted. The **begin end-alternate-and-halt** construct is the same as the above, yet if any one of the alternated procedures completes its execution, *all* "sister" procedures and *all* their sibling procedures are deleted. For example, the scan procedure (Figure 2) is added to the alternated list by fast_flip, but will not terminate upon termination of the alternate construct in shared_coin_flip (Figure 3). It will however be terminated upon termination of the **begin end-alternate-and-halt** construct of terminating_consensus (Figure 6), of which it is a sibling by way of the consensus algorithm.

Notice that the **begin end-alternate** construct is just a coding convenience, used to simplify the complexity analysis and modularize the presentation. It is implemented locally at one process and does not cause spawning of new processes. For all practical purposes, it could be directly converted into sequential code. The resulting constructed algorithm will have the running time of the faster algorithm and the fault-tolerance of the more resilient, though it could be that the different processors do not all finish in the same procedure. For example, in the case of the coin-flip of Figure 3, this could mean that different processors end with different outcomes, depending on which of the interleaved coin-flip operations they completed first.

4 A Fast Write-Scan Primitive

Most of the known consensus algorithms include some type of information gathering stage that requires each processor to perform a read of $(n-1)$ different locations in the shared memory. A fairly simple adversary can exploit the asynchrony of the system to ensure that this stage requires $n-1$ time steps even if there are no faults. However, in this section we show how to implement a scan of a write-once shared *memory_vector* so that each processor obtains the results within expected time $O(\log n)$ even in the case that there are $n^{1-\epsilon}$ faulty processors. This fast behavior is obtained by having processors share the work for the scan. A major difficulty is that, because the processors call the scan asynchronously, the scan that one processor obtains may not be adequate for another process that began its scan later. (Recall that a valid scan must return a value for each process that wrote before that particular scan was called.)

A processor performing a scan of an array needs to collect values written by other processors. The main idea for collecting these values more quickly is to have each participating process record all of the information it has learned about the array in a single location that can be read by any other processor. When one processor reads from anothers location it learns not only that processor's value, but all values that that processor has learned.

In what order should processors read other's information so as to spread it most rapidly? The difficulty here is to define such an ordering that will guarantee rapid spreading of the information in the face of asynchrony and possible faults. Our solution is very simple: each participating processor chooses the next processor to be read *at random*.

The above process can be viewed as the spreading of communicable diseases among the processors. Each processor starts with its own unique disease (the value it wrote to the array being scanned), and each time it reads from another processor, it catches all the diseases that processor has. Proving upper bounds on the expected time of a scan amounts to analyzing the time until everyone catches all diseases. The analysis is complicated by the fact that the processors join each disease-spreading process asynchronously. Furthermore, some of them may spontaneously become faulty. In

fact, achieving the level of fault tolerance that we claim requires a modification in the procedure above. Instead of always reading a processor randomly chosen from among *all* processors, a processor alternately chooses a processor in this manner and then a processor at random from the set of processors whose value (disease) it does not yet have.

Because of the presence of asynchrony and failures, it is neither necessary nor possible to guarantee that a processor always gets a value for every other processor's memory-location. The requirement of a scan is simply that the processor obtains a value for all processors that wrote before the scan began. Thus far, however, we have ignored a crucial issue: since a processor is not personally reading all other processors registers, how can it know that the processors for which it has no value did not write before it started its scan? Define the relation *before* by saying that $before(j,k)$ holds if P_j began its scan before P_k completed its write.If all processors write before scanning then this relation is transitive. The solution now is for each processor to record all *before* relations that it has learned or deduced. Now a processor P_i can terminate its scan as soon as it can deduce $before(i,j)$ for each processor P_j for which it has no value.

A further complication in the operation of the scan occurs because processors may want to scan the same vector more than once. In this case, the *before* relations that hold with respect to one scan of the processor need not hold for later scans. Thus processors must distinguish between different calls to scan by maintaining scan-number counters, readable by all, and by passing information regarding the last known scan-number for each of the other processors.

The time analysis of this algorithm essentially reduces to a careful analysis of the "disease spreading" process described above. This analysis (which will be presented in the full paper) results in the following:

Theorem 4.1 *If all non-faulty processes participate in the scan, then the expected time until each obtains a scan is* $O(\frac{\log^2 n}{\log n - \log f})$.

```
function scan(mem: memory_vector);
  procedure random_update(j : id);
   begin
 1: latest_known_scan_number_i[j] :=
      latest_known_scan_number_j[j];
 2: for k ∈ {1..n}
        do update mem.view_i[k] by any
        mem.view_j[k] ≠ ⊥ od;
 3: if mem.view_i[j] = ⊥ then
 4:   for k ∈ {1..n} do mem.before_i[k, j] :=
        latest_known_scan_number_i[k]
      od fi;
 5: for k, l ∈ {1..n} do mem.before_i[k, l] :=
      max(mem.before_i[k, l], mem.before_j[k, l])
    od;
 6: for k, l ∈ {1..n} do update mem.before_i[k, l]
      based on the transitive closure
      of mem.before_i od;
   end;
  begin
 1: increment latest_known_scan_number_i[i] by 1;
 2: if latest_known_scan_number_i[i] = 1 then
 3:   begin-alternate
 4:     ( repeat
 5:         choose j uniformly from {1..n} − {i}
              do random_update(j) od;
 6:         choose j uniformly from the set of
                processes k such that
                mem.view_i[j] = ⊥
              do random_update(j) od;
 7:     until for every k mem.view_i[k] ≠ ⊥;)
 8:   end-alternate
 9: repeat read ith mem entry
10: until for every k mem.view_i[k] ≠ ⊥ or
      mem.before_i[k] =      ;
11: return mem.view_i;
  end;
```

Figure 2: Fast Scan — Code for P_i.

As mentioned in the introduction, the analysis of this implementation of scan assumes that the shared registers have quadratic size and that computation other than shared memory accesses is negligible.

These assumptions can be eliminated by using an alternative procedure, which works deterministically using a shared binary tree data

structure. The leaves of this tree are the entries of the memory-vector being scanned. Each of the $n-1$ shared variables corresponding to the internal nodes of the tree has a different writer and the entry of the memory-vector belonging to that writer is a leaf in the subtree of the internal node.

The scan is performed by collecting information through the scan tree. The scan algorithm for a process consists of two interleaved procedures. The first is a waiting algorithm that continually checks the children of the internal node controlled by the process to see if they have both been written, and if so, writes the combined information at the internal node. The second is a wait free algorithm that does a depth first search of the scan tree, advancing only from nodes that are not yet written.

As described, this algorithm still requires that each internal node represent a large register, in order to store all of the information that has been passed up. However, we can now take advantage of the fact that whenever a process performs a scan in the consensus algorithm, it does not need to know the distinct entries of the memory-vector. Rather, the process only needs to know which of the two binary values (0 or 1, in the case of a scan of *proposed*[r] and agree or disagree, in the case of *check*[r]) appeared in its scan. Thus, it is only necessary for each internal node to record. the subset of values that appear in the leaves below it. (This is not quite the whole story; a memory-vector scan is also used in the shared-flip procedure of the next section and the information that must be recorded in each node is the *number of 1's* at the leaves of the subtree.)

The main drawback relative to the other implementation is that the expected time for executions with f faults, which is $O((f+1)\log n)$, degrades more rapidly as the number of faults increases.

```
function shared_coin_flip (r: integer);
    begin
        begin-alternate
1:          ⟨ return leader_flip (r)⟩;
        and
2:          ⟨ return fast_flip (r)⟩;
        and
3:          ⟨ return slow_flip (r)⟩;
        end-alternate;
    end;
```

Figure 3: The Shared Coin - Code for P_i.

5 A Fast Joint Coin Flip

Recall from Section 2, that the expected number of rounds to reach a decision is $1/\delta$ where δ is the agreement parameter of the coin. In [4] Aspnes and Herlihy showed how to implement such a shared coin with a constant agreement-parameter, and expected running time in $O(n^3/(n-f))$. This time for implementing the coin is the main bottleneck of their algorithm.

In this section, we give three shared-coin constructions, one trivial one for failure free executions that takes $O(1)$ expected time, one new one which runs in expected time $O(\log n)$ for executions where $f \leq \sqrt[3]{n}$, and a third which achieves the properties of the Aspnes-Herlihy coin with a simplified construction, and runs in $O(\frac{n^3}{n-f})$ expected time for any f. Using an alternated interleaving construct we combine these algorithms to get a single powerful shared global coin enjoying the best of all three algorithms. (See Figure 3. Notice that the shared coin procedure does not terminate until one of the alternate-interleaved return statements is completed.)

The *leader-coin* is obtained by having one pre-designated processor flip its coin and write the result to a shared register. All the other processors repeatedly read this register until the coin value appears. While this coin is only guaranteed to terminate if the designated pro-

cessor is non-faulty, on those executions it takes at most $O(1)$ time and has an agreement parameter 1/2.

The other two coins are motivated by a simple fact from probability theory, which was also used by [4] to construct their coin: For sufficiently large t, in a set of t^2+t independent and unbiased coin flips, the probability that the minority value appears less than $t^2/2$ times is at least 1/2.

The *slow flip* algorithm of Figure 4 is similar in spirit to the coin originally proposed in [4]. The processors flip their individual coins in order to generate a total of n^2 coin flips; the value of the global coin is taken to be the majority value of the coins. To accomplish this, each processor alternates between two steps: 1) flipping a local coin and contributing it to the global collection of coins (Lines 4-5), and 2) checking to see if the n^2 threshold has been reached and terminating with the majority value in that case (Lines 6-7). Due to the asynchrony of the system, it is possible that different processors will end up with different values for the global coin. However, it can be shown that the total number of local coins flipped is at most $n^2 + (n-1)$. Notice that whenever the minority value of the entire set of flips occurs fewer than $n^2/2$ times, every processor will get the same coin value. By the observation of the previous paragraph, this occurs with probability at least 1/2, and thus the algorithm has constant agreement parameter at least 1/4. Furthermore it tolerates up to $n-1$ faulty processors and runs in $O(\frac{n^3}{n-f})$ time. (Using the fast scan this can be reduced to $O(\frac{n^2}{n-f}\log n)$; details are omitted).

In the final *fast flip* algorithm of Figure 5, a processor flips a single coin, and writes it to its location in a shared memory vector (Line 1). Then it repeatedly scans the collection of coins until it sees that at least $n-\sqrt{n}$ of the processors have contributed their coins and at that point it decides on the majority value of those coins it saw (Lines 3-4). We can apply the probabilistic observation to conclude that the minority value will occur less than $(n-\sqrt{n})/2$

```
function slow_flip(r: integer);
  begin
1:   slow_coin.num_flips_i[r] := 0;
2:   slow_coin.num_ones_i[r] := 0;
     repeat
3:     coin := coin_flip;
4:     increment slow_coin.num_flips_i[r] by 1;
5:     increment slow_coin.num_ones_i[r] by coin;
6:     for all j read slow_coin.num_flips_j[r]
         and slow_coin.num_ones_j[r]
       sum respectively into total_flips
         and total_ones;
7:   until total_flips ≥ n^2;
8:   if total_ones/total_flips ≥ 1/2
9:     then return 1
10:    else return 0
     fi;
  end;
```

Figure 4: A Slow Resilient Coin - Code for P_i.

times with probability at least 1/4 in which case all processors will obtain the majority value as their shared-flip. Of course, if there are more than $\sqrt{n}$ faulty processors then it is possible that no processor will complete the algorithm. However, in the case that the number of faulty processors is at most $\sqrt{n}$, all non-faulty processors will complete the algorithm using up one unit of time to toss the individual coins (and perform all but the last completed scan) then the time for the last scan. The result is an algorithm that is resilient for up to $\sqrt{n}$ faults and runs in expected time $O(\log n)$ in that case.

Finally, let us consider the expected running time of the composite coin. The interleaving operation effectively slows the time of each of the component procedures by a factor of three; but since the processor stops as soon as one of three procedures terminates, for each number of faults the running time can be bounded by the time of the procedure that terminated first process. The agreement parameter of the composite coin is easily seen to be at least the product of the agreement parameters of the individual coins. We summarize the property of

```
function fast_flip(r: integer);
  begin
1:   write(fast_coin[r],coin_flip);
2:   repeat coin_view := scan(fast_coin[r])
3:   until coin_view contains at least
       n − √n non-⊥ values;
4:   return the majority non-⊥ value
       in coin_view;
  end;
```

Figure 5: A Fast Coin Flip - Code for P_i.

the joint coin with:

Theorem 5.1 *The joint-coin implements a shared-flip with constant agreement parameter. The expected running time for any number of faults less than n is* $O(\frac{n^3}{n-f})$. *In any execution where there are no faults, the expected running time is* $O(1)$. *In any execution in which there are at most* $\sqrt{n}$ *faults, the expected running time is* $O(\log n)$.

6 Ensuring Fast Termination

Our consensus algorithm is obtained from the main consensus algorithm presented in Section 2 by using the fast implementations of the scan and shared_coin_flip given in the last two sections. There is one difficulty; the implementations of these primitives require each processor to continue participating in the work even after it has received its own desired output from the function; this shared work is necessary to obtain the desired time bounds. As a result, each processor is alternating between the steps of the main part of the consensus algorithm and work of various scans and shared-coins in which it is still participating. Thus when it reaches a decision, its consensus program halts, but the other interleaved work must continue, potentially forever. Furthermore, if the processor simply stops working on the scans that are still active, then its absence could delay the completion of other processors, resulting in a high time complexity. To solve this problem we add a new *memory-vector* called *decision_value*. Upon reaching a decision, each processor writes its value in this vector before halting. Now, we embed the main consensus algorithm in a new program called terminating consensus (see Figure 6) that simply alternates the main consensus algorithm with an algorithm that monitors the *decision_value* vector, using an **begin end-alternate-and-halt** construct. If a processor ever sees that some other process has reached a decision, it can safely decide on that value and halt. Furthermore, by the properties of the scan, once at least one processor has written a *decision_value*, the expected time until all non-faulty processors will see this value is bounded above by a constant multiple of the expected time to complete a scan.

Let us now consider the time complexity of the algorithm of Figure 1. Since the agreement parameter of the shared-coin is constant, the expected number of rounds is constant. Essentially each round consists of a constant number of writes, two scan operations and one shared-coin flip; Putting together the properties of the various parts of the consensus algorithm we get:

Theorem 6.1 *The algorithm terminating consensus satisfies the correctness, validity and termination properties. Furthermore, on any execution with fewer than* $O(\sqrt{n})$ *failures the expected memory based time until all non-faulty processors reach a decision is* $O(\log n)$ *and the expected time is* $O(n \log n)$. *Otherwise, the expected time until all non-faulty processors reach a decision is* $O(\frac{n^3}{n-f})$.

7 Modularly Speeding-Up Resilient Algorithms

In this section we present a constructive proof that any decision problem that has a wait-free or expected wait-free solution, has an expected wait-free solution that takes *only* $O(\log n)$ expected time in normal executions.

```
function terminating_consensus (v: input_value);
  begin
   begin-alternate
1:   ⟨ write(decision_value[r], consensus(v);)
   and
     ⟨ repeat
2:     choose j uniformly from {1..n} − {i}
3:     decision := (j'th position of decision_value[r])
4:     until decision
            is not ⊥;
5:   write(decision_value[r], decision); ⟩
   end-alternate
  end;
```

Figure 6: Terminating — Code for P_i.

Theorem 7.1 *For any decision problem P, given any wait-free or expected wait-free solution algorithm $A(P)$ as a black box, one can modularly generate an expected wait-free algorithm with the same worst-case time complexity, that runs in only $O(\log n)$ expected time in failure-free executions.*

For lack of space we only outline the method on which the proof is based. Given $A(P)$, it is a rather straightforward task to design a non-resilient "waiting" algorithm to solve P. As with the tree scan of Section 4, we use a binary tree whose leaves are the input variables to pass up the values through the tree. A processor responsible for an internal node waits for both children to be written and passes up the values. Each processor waits to read the root's output which is the set of all input values, locally simulating $A(P)$ on the inputs to get the output decision value [3]. This algorithm takes at most $O(\log n)$ expected time in executions in which no failures occur. We can now perform an alternated-interleaving execution of $A(P)$; and the waiting algorithm described above, that is, each processor alternates between talking a step of each, returning the value of the first one completed. Though the new protocol takes $O(\log n)$ time in the failure free executions and has all the resiliency and worst case properties of $A(P)$, there is one major problem: it is possible for some processors to terminate with the decision of $A(P)$, while others terminate with the possibly different decision of the waiting protocol. The solution is to have each process, upon completion of the interleaved phase, participate in a fast consensus algorithm, with the input value to consensus being the output of the interleaving phase. All processors will thus have the same output within an added logarithmic expected time, implying the desired result.

Based on Theorem 7.1, we can derive an $O(\log n)$ expected wait-free algorithm to solve the *approximate agreement* problem, using any simple wait-free solution to the problem as a black-box (this compares with the optimally fast yet intricate deterministic wait-free solution of [7]). The Theorem also implies a fast solution to *multi-valued* fault-tolerant consensus. As a black box one could use the simple exponential-time multi-valued consensus of [1] (or for better performance the above consensus algorithm with the coin flip operation replaced by a uniform selection among up to n values).

8 Acknowledgements

We wish to thank Danny Dolev and Daphne Koller for several very helpful discussions.

[3] Note that if $A(P)$ is a randomized algorithm, all processors can use the same predefined set of coin flips. Also, in practice, it is enough to know the input/output relation P instead of using the black-box $A(P)$.

References

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

[2] 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.

[3] E. Arjomandi, M. Fischer and N. Lynch, "Efficiency of Synchronous Versus Asynchronous Distributed Systems," *Journal of the ACM*, Vol. 30, No. 3 (1983), pp. 449–456.

[4] J. Aspnes and M. Herlihy, "Fast Randomized consensus Using Shared Memory," *Journal of Algorithms*, September 1990, to appear.

[5] J. Aspnes, "Time- and Space-Efficient Randomized Consensus," to appear in *the 9th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 325–332, August 1990.

[6] H. Attiya, D. Dolev, and N. Shavit, "Bounded Polynomial Randomized Consensus," Proc. 8th ACM Symp. on Principles of Distributed Computing, pp. i281–294, August 1989.

[7] H. Attiya, N. Lynch, and N. Shavit, "Are Wait-Free Algorithms Fast?" *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pp. 55–64, October 1990.

[8] S. Chaudhuri, "Agreement is Harder Than Consensus: Set Consensus Problems in Totally Asynchronous Systems", Proc. 9th ACM Symp. on Principles of Distributed Computing, 1990, pp. 311-324.

[9] 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.

[10] 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.

[11] B. Coan and C. Dwork, "Simultaneity is Harder than Agreement", *Proc. 5th IEEE Symposium on Reliability in Distributed Software and Database Systems*, 1986.

[12] D. Dolev, C. Dwork, and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus," J.cACM 34, 1987, pp. 77–97.

[13] C. Dwork and Y. Moses, "Knowledge and Common Knowledge in a Byzantine Environment: Crash Failures," to appear in *Information and Computation.*

[14] M. Fischer, N. Lynch and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Processor," *Journal of the ACM*, Vol. 32, No. 2 (1985), pp. 374–382.

[15] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Processor," J.cACM 32, 1985, pp. 374–382.

[16] V. Hadzilacos and J. Y. Halpern, "Message- and bit-optimal protocols for Byzantine Agreement," unpublished manuscript, 1990.

[17] M. P. Herlihy, "Wait Free Implementations of Concurrent Objects," Proc. 7th ACM Symp. on Principles of Distributed Computing, 1988, pp. 276–290.

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

[19] B. Lampson, "Hints for Computer System Design", in *Proc. 9th ACM Symposium on Operating Systems Principles*, 1983, pp. 33–48.

[20] M. Loui and H. Abu-Amara, "Memory Requirements for Agreement Among Unreliable Asynchronous Processes," *Advances in Computing Research*, Vol. 4, JAI Press, Inc., 1987, 163-183.

[21] N. Lynch and M. Fischer, "On Describing the Behavior and Implementation of Distributed Systems," *Theoretical Computer Science*, Vol. 13, No. 1 (January 1981), pp. 17-43.

[22] Y. Moses and M. Tuttle, "Programming Simultaneous Actions using Common Knowledge," *Algoritmica*, Vol. 3, 1988, pp. 121–169.

[23] S. Plotkin, "Sticky Bits and the Universality of Consensus," Proc. 8th ACM Symp. on Principles of Distributed Computing, August 1989.

[24] M. Rabin, "Randomized Byzantine Generals," *Proc. 24th IEEE Symp. on Foundations of Computer Science*, pp. 403–409, October 1983.