Math. Systems Theory **26**, 21–39 (1993)

Mathematical Systems Theory

A Partial Equivalence Between Shared-Memory and Message-Passing in an Asynchronous Fail-Stop Distributed Environment*

Amotz Bar-Noy[1] and Danny Dolev[2]

[1]IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA

[2]IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, USA and
Computer Science Department, The Hebrew University, Jerusalem, Israel

Abstract. This paper presents a schematic algorithm for distributed systems. This schematic algorithm uses a "black-box" procedure for communication, the output of which must meet two requirements: a global-order requirement and a deadlock-free requirement. This algorithm is valid in any distributed system model that can provide such a communication procedure that complies with these requirements. Two such models exist in an asynchronous fail-stop environment: one in the shared-memory model and one in the message-passing model. The implementation of the block-box procedure in these models enables us to translate existing algorithms between the two models whenever these algorithms are based on the schematic algorithm.

We demonstrate this idea in two ways. First, we present a randomized algorithm for the consensus problem in the message-passing model based on the algorithm of Aspnes and Herlihy [AH] in the shared-memory model. This solution is the fastest known randomized algorithm that solves the consensus problem against a strong fail-stop adversary with one-half resiliency. Second, we solve the processor renaming problem in the shared-memory model based on the solution of Attiya *et al.* [ABD$^+$] in the message-passing model. The existence of the solution to the renaming problem should be contrasted with the impossibility result for the consensus problem in the shared-memory model [CIL], [DDS], [LA].

*A preliminary version of this paper, "Shared-Memory vs. Message-Passing in an Asynchronous Distributed Environment," appeared in *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pp. 307–318, 1989. Part of this work was done while A. Bar-Noy visited the Computer Science Department, Stanford University, Stanford, CA 94305, USA, and his research was supported in part by a Weizmann fellowship, by Contract ONR N00014-88-K-0166, and by a grant of Stanford's Center for Integrated Systems.

1. Introduction

Two major processor intercommunication models in distributed systems have attracted much attention and study: the *message-passing* model and the *shared-memory* model. In the message-passing model, processors are represented by vertices of a *network* and messages are sent along links. In the shared-memory model, processors share *registers* through which they communicate by performing read and write operations.

In particular, the *asynchronous* environment and the *fail-stop* environment have been the underlying assumptions of many recent papers. Researchers tried to understand distributed systems where processors operate at different rates and do not share a unique global clock. They also tried to understand distributed systems where some of the processors may suddenly fail and stop operating (*crash*). The main difficulty in an asynchronous and fail-stop environment is how to distinguish between slow processors and crashed processors. The distinction between these two modes of operation is the challenge in solving *coordination* problems. In such problems a set of processors are asked to make a decision depending upon some initial input that each one of them possesses. The Agreement Problem [PSL] and the Leader Election Problem [GHS] are among the famous coordination problems studied in recent decades.

Unfortunately, since the shared-memory model and the message-passing model are not equivalent, many coordination problems were solved for each model separately. Therefore, finding ways to translate existing algorithms from one model to the other is an intriguing question in distributed systems. This paper provides a way to translate certain types of algorithms between these two models in an asynchronous and fail-stop environment. The goal is to find an equivalence that preserves the complexity measures: time, the size of messages or shared registers, local memory and local computation. In certain cases, proving that the existence of a solution in one model implies its existence in the other is important by itself.

We present a *schematic algorithm* for distributed systems. This algorithm uses a *black-box* procedure for communication. The collection of all outputs of this procedure, no matter which processor invoked the procedure, must meet two requirements: a *global-order* requirement and a *deadlock-free* requirement. Any proof of correctness assuming only these two requirements regarding the outputs is valid in any distributed system model that can provide a communication procedure that complies with these requirements. We then introduce two such communication procedures in an asynchronous fail-stop environment: one in the shared-memory model where the communication is via *wait-free atomic* read and write registers and one in the message-passing model where the communication is via a *fully connected* network and the *resiliency* is one-half. Applying these procedures enables us to translate existing algorithms between the two models whenever these algorithms are based on the schematic algorithm. (Note that Chor and Moscovici [CM] have demonstrated that for certain types of problems the above two models are not equivalent. There are problems that can be solved in the message-passing model with resiliency one-half but have no wait-free solution in the shared-memory model and vice versa.)

Two examples of such algorithms exist. One is the solution of Aspnes and Herlihy [AH] to the *consensus problem* in the shared-memory model and one is the solution of Attiya *et al.* [ABD$^+$] to the *renaming problem* in the message-passing model. These solutions assume only the global-order requirement and the deadlock-free requirement and, consequently, these solutions are valid in both models. The randomized solution for the consensus problem in the message-passing model is the fastest known randomized algorithm for solving this problem with resiliency one-half against the strongest possible adversary in a fail-stop model. The time complexity of this solution is $O(n^2)$ expected number of rounds where n is the number of processors. This improves the result of Ben-Or [B] of $O(2^n)$ expected number of rounds. The solution for the renaming problem in the shared-memory model is the first deterministic wait-free solution to a coordination problem using atomic registers. Its existence should be contrasted with the impossibility result for the consensus problem in this model [CIL], [DDS], [LA].

The rest of the papere is organized as follows. In Section 2 we describe the two models considered in this paper. The schematic algorithm is presented in Section 3 and the two communication procedures are presented in Section 4. Our two applications for the consensus problem and the renaming problem appear in Sections 5 and 6 respectively.

2. The Shared-Memory and the Message-Passing Models

2.1 The Shared-Memory Model

The shared-memory model is defined as follows (for more details see [AH]). There are n processors that communicate by reading and writing into *registers*. The registers are *atomic single-writer many-reader registers*. Each processor has an assigned register and is the only one that can write to it. However, all processors can read all registers. The nonlocal atomic operations a processor can execute without interference are READ and WRITE on a single register. Thus, once a processor begins each of these operations it can complete it without the need to wait for any other processor (see [L]). Intuitively, this property implies that a reader need not wait for a writer to complete a read operation. Thus, even if the writer crashes, the reader can complete its read operation. Similarly, a writer need not wait for readers to complete their operations.

This requirement can be extended to any action a processor performs. In this paper we focus only on *wait-free* protocols, where a processor is able to complete any action or set of actions without waiting for the completion of other actions at any other processor. This models the asynchrony of the system and the fact that even $n - 1$ processors may crash. However, a processor cannot quit after making its decision before all other processors make their decision. It cannot erase the content of its register, otherwise some processors may assume that other processors that had already made their decisions are crashed processors and this may lead to inconsistent decisions in many coordination problems.

This shared-memory model has been considered in many papers. Let us now summarize some results that are relevant to our paper. In various papers

(see [CIL], [DDS], and [LA]) it was proven that there is no determinstic wait-free solution for the consensus problem when the atomic operations are read and write. Atomic operations that are sufficient to solve the consensus problem are explored in [H]. In several papers (see [A], [AH], [CIL], and [ADS]) randomized algorithms are represented for the consensus problem.

2.2. *The Message-Passing Model*

We define the message-passing model as follows (for more details see [ABD$^+$]). There are n processors that are connected by a complete network. We assume that when a processor sends a message, it broadcasts the same message to all other processors. The fail-stop environment is characterized by a parameter t ($t < n$) which is an upper bound for the number of processors that may crash. The asynchrony is represented by a game between an *adversary* and a given algorithm. The goal of the adversary is the failure of the algorithm.

This paper considers the following strong adversary in an asynchronous and fail-stop environment. The adversary instructs the processors when to operate and when to stop. However, it can stop up to t processors. Whenever a processor waits for messages, it cannot expect to receive more than $n - t$ messages from different processors since the other t messages may belong to crashed processors. Messages can arrive out of order as the adversary has the power to dictate their delivery time. The adversary can plan its strategy by observing the messages to be sent and the internal state of the processors. There are no cryptographic assumptions, or other limitations on the power of the adversary. It is only prevented from changing messages, changing the steps processors take, and predicting the outcome of coin flips in randomized algorithms. The number t, the upper bound for the number of processors that may crash, is called the *resilience* parameter of the system. (Traditionally, the parameter t was an upper bound on the number of *faulty* processors, which do not follow their algorithm.)

Following the proof in [ABD$^+$] it is easy to see that, for many coordination problems, if $n \leq 2t$, there is no algorithm that can prevail over such an adversary. Because the adversary can force two groups of at most t processors to operate separately by suspending all the messages between the two groups and this may lead to inconsistent actions by the two groups. Furthermore, by similar reasoning, at least $2t + 1$ processors, even if they have already made their own decisions, must continue sending and receiving messages in order to help other processors make their decision. In this paper, t is set to its maximal possible value for many coordination problems, i.e., $n = 2t + 1$.

Let us now summarize some results in this message-passing model that are relevant to our paper. Fischer *et al.* [FLP] proved that the consensus problem is not solvable deterministically against such an adversary even when $t = 1$. Yet, it is possible to overcome the adversary with a randomized algorithm [B], [CMS], and [DSS]. The result of [FLP] was generalized in the papers [DDS] and [DLS] where the level of synchrony needed to solve the consensus problem is studied. Finally, some papers (e.g., [ABD$^+$], [BMZ], and [BW]) identify certain problems that are solvable determinstically in the presence of such an adversary.

3. The Schematic Algorithm

In this section we present a schematic algorithm for an asynchronous distributed system (Figure 1). The only way processors communicate between themselves is by calling Procedures COMMUNICATE and HELP-COMMUNICATE. The rest of the code is local and is independent of the specific model in which the schematic algorithm is invoked. However, the schematic algorithm demands two requirements from the outputs of Procedure COMMUNICATE, called the *global-order requirement* and the *deadlock-free requirement*. The schematic algorithm is used as follows. Let $\mathscr{A}$ be a distributed algorithm, the code of which follows the code of the schematic algorithm (Figure 1). Suppose that the global-order requirement and the deadlock-free requirement are the only properties of Procedure COMMUNICATE used in the proof of $\mathscr{A}$. Then the proof is valid in any model that can implement Procedure COMMUNICATE, as long as the implementation satisfies these two requirerments.

A processor p, that executes Algorithm SCHEMATIC, gets some **Input** and needs to produce some **Output**. We say that processor p *decides* at the moment p is able to produce the output. Let I be the set of all the local variables of processor p, where some of the variables get their initial value from the input. A subset of I, or some function of I, is communicated with other processors in the system. We call these variables the *communication variables* of p. Processor p collects the communication variables of other processors. For that purpose, p maintains a vector X of length n. The entry $X(q)$ of the vector is the most recent value of the communication variables of processor q that p knows about. The entry $X(p)$ is the value of the communication variables of p itself and it contains its most updated value. We call this vector the *communication vector* of processor p. Processor p works in *rounds*. In each round, p first communicates with the other processors and thereby updates its communication vector. Then p updates its local variables (I) and updates the appropriate entry in its communication vector ($X(p)$). Processor p continues to the next round if it has not yet decided. When p decides and the

```
Algorithm SCHEMATIC(Input, Output);
var  I: set of variables;
     X: vector of length n;
begin
  I:= some function of Input;
  for each processor q do X(q):= a null value;
  while decision is not made do
    X(p):= some function of I;
    COMMUNICATE(X);
    I:= some function of X;
  end-while;
  Output:= some function of I;
  HELP-COMMUNICATE(X);
end;
```

Fig. 1. The schematic algorithm for processor p.

output is produced, processor p calls Procedure HELP-COMMUNICATE. As mentioned in the previous section, in both models p cannot stop completely until everybody has made a decision. In the shared-memory model, p cannot erase the contents of its register, and in the message-passing model, p must actively communicate with other processors.

We now define the global-order requirement. First, we assume the following *local-order assumption*:

Assumption 3.1. *Let the values of the communication variables of all processors be taken from a partially ordered collection $\mathcal{J}$ containing a null value which is less than all the other values. If, in times $t_1 < t_2$, processor p held the communication variables value J_1 and J_2 respectively, then $J_1 \preceq J_2$. Moreover, every other processor can compute $\preceq$.*

Note that this can be satisfied by a time-stamp system. An unbounded time-stamp system can be implemented by adding an unbounded counter to the communication variables; this counter is incremented for any new version of these variables. In the shared-memory model, previous results show ways to use bounded time-stamp system [II], [DS].

Now, a partial order on the communication vectors can be defined. Recall that if processor p holds the communication vector X, then $X(q)$ is a copy of the most recent communication variable of processor q known to p.

Definition 3.1. Let X and X' be two communication vectors held by processors p and q, respectively. Then $X \preceq X'$ iff, for every $1 \leq k \leq n$, $X(k) \preceq X'(k)$.

In other words, X precedes X' iff in all its entries X precedes X' according to the order defined for $\mathcal{J}$ in Assumption 3.1.

Definition 3.2. The Global-Order Requirement. All the outputs of Procedure COMMUNICATE must form a linear order under $\preceq$, no matter which processor invoked the procedure.

In any run of Algorithm SCHEMATIC by all the processors, in which the global-order requirement is met, we define the following.

Definition 3.3. A stable vector is a vector that was returned as an output of Procedure COMMUNICATE. Denote by $X_1 X_2, \ldots$ the sequence of all stable vectors output by any call to Procedure COMMUNICATE, and let X_0 be the vector with all entries equal to the null value.

A possible undesirable outcome of the global-order requirment is deadlock. It might be the case that the processors will communicate forever because they cannot satisfy this requirement. The second requirement avoids such situations.

Definition 3.4. The Deadlock-Free Requirement. If any nonfaulty, undecided

processor is executing Procedure COMMUNICATE, then eventually some processor will output a stable vector.

Note that this definition permits starvation. However, in many coordination problems, when there is a need only for one decision, a deadlock-free property implies a nonstarvation property.

4. The Communication Procedures

In this section we provide four communication procedures (Figures 2 and 3), two in each of the models defined in Section 2. In the shared-memory model these procedures are called SM-COMMUNICATE and SM-HELP-COMMUNICATE, and in the message-passing model they are called MP-COMMUNICATE and MP-HELP-COMMUNICATE.

The input for all four procedures is a communication vector of length n (which contains the most updated version of the communication variables of other processors known to the processor that calls the procedure). The first time Procedures SM-COMMUNICATE and MP-COMMUNICATE are invoked by processor p, the input vector is the initial vector. In the initial vector, all the entries are set to their null value except for p's entry, which is set to the value of p's communication variables. In later invocations, the input vector is the current vector known to processor p.

In Procedure SM-COMMUNICATE, processor p *scans* all the n registers and copies their values into a temporary vector (Y). If the vectors that result from two consecutive scans are identical, then this vector is the output and the procedure

```
Procedure SM-COMMUNICATE(X);
Input   X: a communication vector of length n;
Output  X: a communication vector of length n;
var     q: a processor name;
        Y: a vector of length n;
begin
  R_p := WRITE(X(p));
  repeat
    for q∈{1,...,n} do Y(q) := READ(R_q);
    if X = Y then return (X);
             else X := Y;
  end-repeat
end;

Procedure SM-HELP-COMMUNICATE(X);
Input   X: a communication vector of length n;
begin
  while not all processors decided do not erase the content of R_p;
end;
```

Fig. 2. The shared-memory communication procedures for processor p. The atomic register of p is denoted by $\mathscr{R}_p$.

```
Procedure MP-COMMUNICATE(X);
Input   X: a communication vector of length n;
Output  X: a communication vector of length n;
var     c: a counter;
        q: a processor name;
        Y: a vector of length n;
begin
1. BROADCAST(X);
   c:= 1;
2. upon receiving Y from q do
     if Y ≺ X then goto 2;
     else if Y = X then begin
                          c := c + 1;
                          if c < n − t then goto 2;
                                       else return (X);
                        end;
     else if Y ⊀ X then begin
                          for all j s.t. X(j) ≺ Y(j) do X(j) := Y(j);
                          goto 1;
                        end;
end;

Procedure MP-HELP-COMMUNICATE(X);
Input   X:: a communication vector of length n;
var     q: a processor name;
        Y: a vector of length n;
begin
  while not all processors decided do
    upon receiving Y from q do
      if Y ⪯ X then SEND(X) to q;
      else if Y ⊀ X then begin
        for all j s.t. X(j) ≺ Y(j) do X(j) := Y(j);
        BROADCAST(X);
      end;
end;
```

Fig. 3. The message-passing communication procedures for processor p.

terminates. Else $X := Y$ and the n registers are scanned again. In Procedure SM-HELP-COMMUNICATE, processor p does not erase the content of its register until all other processors have made their decision. (Note that Procedure SM-HELP-COMMUNICATE does not need the input X. However, we leave it like this for the sake of unique presentation of the schematic algorithm.) The register of processor p is denoted by $\mathscr{R}_p$ and an atomic read or write of a value v into or from register $\mathscr{R}_p$ is denoted by $v := \text{READ}(\mathscr{R}_p)$ or $\mathscr{R}_p := \text{WRITE}(v)$, respectively.

In Procedure MP-COMMUNICATE, processor p broadcasts its communication vector to other processors and updates its own communication vector according to the messages it receives. Since processor p is guaranteed to receive at most $n - t$ messages from different processors, it learns about the values of the communication variables of the rest of the t processors, if these variables exist, through other processors. The way this is done is as follows. During the run of

Procedure MP-COMMUNICATE, processor p has a candidate vector. The vector becomes the output when processor p receives $n - t - 1$ vectors identical to its own candidate. If processor p receives a vector that is more recent in one of its entries according to $\preceq$, it updates all such entries in its candidate vector and resumes collecting identical vectors. In Procedure MP-HELP-COMMUNICATE, processor p continues to update its communication vector and to broadcast more updated versions of its communication vector until all other processors made their decision. The command BROADCAST(X) stands for sending the message X to all other processors.

We note that in both models it is often impossible to determine whether all the processors have made their decision. Obviously, running Procedure SM-HELP-COMMUNICATE and Procedure MP-HELP-COMMUNICATE forever is enough. However, if there exists a way to know that all the processors have decided, then the processors may quit.

We need to prove that these procedures comply with the global-order requirement and the deadlock-free requirement.

Lemma 4.1. *Suppose that X_1 and X_2 were the communication vectors of processor p in times $t_1 < t_2$. Then $X_1 \preceq X_2$.*

Proof. In both Procedures SM-COMMUNICATE and MP-COMMUNICATE, a processor updates an entry in its communication vector only by more updated information (according to $\preceq$). Therefore, all the vectors p held are totally ordered under $\preceq$. □

Lemma 4.1 implies that both procedures meet the global-order requirement.

Lemma 4.2 (Global-Order). *Let X and X' be two output vectors held by processors p and q, respectively. Then either $X \preceq X'$ or $X' \preceq X$.*

Proof of the Shared-Memory Model. Assume to the contrary that the lemma is false. Then there exist l and k such that $X(l) \prec X'(l)$ and $X'(k) \prec X'(k)$. Denote by t_1, t_2, t_3, t_4 the times that p read $\mathcal{R}_l$ in the first scan of X, read $\mathcal{R}_k$ in the first scan of X, read $\mathcal{R}_l$ in the second scan of X, and read $\mathcal{R}_k$ in the second scan of X, respectively. Similarly define s_1, s_2, s_3, s_4 for q.

By definition, both t_1 and t_2 are smaller than both t_3 and t_4 and both s_1 and s_2 are smaller than both s_3 and s_4. Since $X(l) \prec X'(l)$, it follows that q, in time s_1, found in $\mathcal{R}_l$ a more updated version of processor l's communication variables than the version p found in time t_3. Consequently, $t_3 < s_1$. Similarly, since $X'(k) \prec X(k)$ it follows that $s_4 < t_2$. Combining the two yields $t_3 < t_2$; a contradiction.

Note that the proof holds even when p and q read the registers $\mathcal{R}_l$ and $\mathcal{R}_k$ not in the same order. This is true since the proof is valid independent of the order between the two times in each of the following four pairs: t_1 and t_2, s_1 and s_2, t_3 and t_4, and s_3 and s_4. Moreover, a processor may read the n registers in two different scans in a different order and the proof still holds. □

Proof for the Message-Passing Model. Both processors p and q had $n-t$ identical vectors for X and X'. Since $n > 2t$ there exists a processor k, the vector of which belongs to both sets of vectors. By Lemma 4.1 the vectors held by any processor are totally ordered and therefore either $X \preceq X'$ or $X' \preceq X$. □

The next lemma shows that both procedures meet the deadlock-free requirement.

Lemma 4.3 (Deadlock-Free). *Suppose that some set of nonfaulty processors, $\mathscr{P}$, have not yet decided. Let X_j be the last stable vector output by any processor. Then eventually one of the processors in $\mathscr{P}$ will output X_{j+1}.*

Proof for the Shared-Memory Model. All the nonfaulty processors that have decided will never change the content of their register. All the other nonfaulty processors will eventually invoke Procedure SM-COMMUNICATE and execute its first line. After that point none of them will change the content of its register unless it outputs a stable vector and it has not yet decided. Thereafter, the first processor in $\mathscr{P}$ to complete two scans will output a stable vector. □

Proof for the Message-Passing Model. A nonfaulty processor p that has already decided will never change the value of $X(p)$ in its communication vector. A nonfaulty processor p that has not yet decided will eventually invoke Procedure MP-COMMUNICATE and will not change the value of $X(p)$ in its communication vector unless it outputs a stable vector. Therefore, eventually, for each nonfaulty processor p, the value of $X(p)$ in its communication vector is fixed.

Recall that processors that have decided, invoke MP-HELP-COMMUNICATE. In any case, at least $n-t$ processors participate in the process of sending and receiving messages. These $n-t$ processors will eventually get each other's vector and will have identical vectors. If p has decided already and outputs a stable vector, it will not change $X(p)$ in its communication vector. Therefore, necessarily one of the undecided nonfaulty processors will output a stable vector. □

5. The Consensus Problem

Definition 5.1. In the *consensus* problem, each processor has an initial binary value. The processors should decide on a binary value under the following requirements:

1. No two processors decide on different values.
2. The decision value is some processor's initial value.

This problem is a very basic coordination problem. A number of papers solve this problem and the related Byzantine Generals problem in different models (see [Fi] for a survey of early work).

Aspnes and Herlihy [AH] described a randomized algorithm for the consensus

problem in the shared-memory model. Their algorithm is wait-free and it uses a *weak shared coin* for the random choices. The expected number of rounds for the coin procedure and for the consensus algorithm is $O(n^2)$. It happens that their consensus algorithm and the procedure that produces the weak shared coin can be based on the schematic algorithm. Therefore, their solution can also be applied to the message-passing model described in Section 2.2.

In the message-passing model, this result improves the best published result of Ben-Or [B], the complexity of which is $O(2^n)$ expected number of rounds. (An unpublished result of Feldman [Fe] states that if $n > 4t$, then there is an algorithm with a constant expected number of rounds.)

5.1. *The Weak Shared Coin*

In this section we describe the coin procedure of [AH] in the framework of Algorithm SCHEMATIC. Aspnes and Herlihy [AH] define their coin using an abstraction for counters. However, it is not hard to verify that our presentation is equivalent. They use a weak shared coin which is defined as follows.

Definition 5.2. For $\alpha > 1$, a weak shared coin has the following two properties:

1. A processor flips $\varepsilon \in \{0, 1\}$ with probability at most $(\alpha + 1)/2\alpha$.
2. With probability greater than $(\alpha - 1)/2\alpha$ all the processors flip the same value.

The parameter α is called the bias parameter.

The idea underlying Procedure COIN is as follows. The processors, together, perform a *random walk* of an abstract *cursor* on the line. The cursor initially points to zero, and each processor can move the cursor one unit to the right or to the left. A processor flips 1 or 0 if the cursor points to the right of αn or to the left of $-\alpha n$, respectively, where α is the bias parameter. Intuitively, the only way the adversary can bias the outcome of the coin toward some $\varepsilon \in \{0, 1\}$, is by preventing as many processors as possible that try to move the cursor to the other direction to do so. In the shared-memory model the adversary can delay the move of $n - 1$ processors and in the message-passing model it can delay the move of t processors. However, in this case the remaining processors continue the random walk. This limits the influence of the adversay on the final outcome of the coins being flipped.

The code for Procedure COIN appears in Figure 4. The code is described for processor p that flips the *coinnumber* coin and computes the weak shared coin *coinvalue* with the bias parameter α. The roll of *coinnumber* is to distinguish between different calls to Procedure COIN. The local function *flip* returns one of the values $+1$ or -1, each with probability 1/2. The local variable *round* generates the local order required by the local-order assumption (Assumption 3.1). The value of the variable *localwalk* is the sum of all the local coins p has flipped. Each entry of the communication vector is composed of three fields associated with

```
Procedure COIN (coinnumber, α, coinvalue);
Input    coinnumber: a counter;
         α: a bias parameter;
Output   coinvalue: 1 or −1;
var      localwalk, globalwalk: integers;
         round: a counter;
         flip: a real coin;
         X: a communication vector;
         (X.coinnumber, X.round, X.localwalk): the three fields in each entry of X;
begin
  round := localwalk := globalwalk := 0;
  for each processor q do X(q) := (coinnumber, 0, 0);
  while −αn ≤ globalwalk ≤ αn do
    round := round + 1;
    (* increments or decrements walk with probability 1/2 *)
    localwalk := localwalk + flip(+1, −1);
    X(p) := (coinnumber, round, localwalk);
    COMMUNICATE(X);
    globalwalk := Σ_{j=1}^{n} X(j).localwalk;
  end-while
  if globalwalk > αn then coinvalue := 1;
  if globalwalk < −αn then coinvalue := 0;
  HELP-COMMUNICATE(X);
end;
```

Fig. 4. Procedure COIN for processor p with bias parameter α.

the communication variables: *coinnumber*, *round*, and *localwalk*. The value of the variable *globalwalk* is the sum of all the *localwalk* fields in p's communication vector. Procedure COMMUNICATE is either Procedure SM-COMMUNICATE or Procedure MP-COMMUNICATE and Procedure HELP-COMMUNICATE is either Procedure SM-HELP-COMMUNICATE or Procedure MP-HELP-COMMUNICATE.

At this stage, we prove that Procedure COIN indeed produces a weak shared coin. We state the lemmas needed and omit the proofs, as they are similar to their counterpart proofs in [AH] (Lemma 15 through Theorem 19).

Denote by H and T the number of 1 and -1 generated by all the processors in some run of Procedure COIN. The global-order lemma (Lemma 4.2) implies Lemma 5.1 which yields Corollaries 5.1 and 5.2.

Lemma 5.1. *Let $1 < \beta$ be a real number. If, for some processor, $H - T > \beta n$ (resp. $T - H > \beta n$), then at that time $H - T \geq (\beta - 1)n$ (resp. $T - H \geq (\beta - 1)n$).*

Corollary 5.1. *If $H - T > (\alpha + 1)n$ (resp. $T - H > (\alpha + 1)n$), then all remaining processors eventually flip 1 (resp. 0).*

Corollary 5.2. *If some processor flips 1 (resp. 0), then at that time $H - T \geq (\alpha - 1)n$ (resp. $T - H \geq (\alpha - 1)n$).*

The next two lemmas show the two properties of the weak shared coin (see Definition 5.2).

Lemma 5.2. *The adversary can force a processor to flip ε with probability at most* $(\alpha + 1)/2\alpha$.

Lemma 5.3. *With probability greater than* $(\alpha - 1)/2\alpha$ *all the processors flip the same value.*

The complexity of this procedure is stated in the next lemma. It can be proved based on the deadlock-free lemma (Lemma 4.3).

Lemma 5.4. *The expected number of rounds of Procedure* COIN *is* $O((\alpha + 1)^2 n^2)$.

Combining Lemmas 5.2, 5.3, and 5.4 together yields the following theorem.

Theorem 5.1. *Procedure* COIN *produces a weak shared coin with* $O((\alpha + 1)^2 n^2)$ *expected number of rounds.*

5.2. The Consensus Algorithm

In this section we describe the consensus algorithm of [AH] in the framework of Algorithm SCHEMATIC. Again, our presentation is not the same as that of [AH] but it is not hard to verify that both presentations are equivalent.

Algorithm CONSENSUS for processor p with initial value v, bias parameter α for coins, and a decision value *cons* appears in Figure 5. In the first round, processor p suggests its input value as the decision value. In later rounds, p either decides on this value or suggests some value again according to the following rules. If, based on p's knowledge, its round is the maximal round and the rounds of all the processors that *disagree* with p's value *trail* by at least two, then p decides on its value. Else, if all the processors with maximum round *agree* on the same value, ε, then p *adopts* ε and suggests it in its next round. Otherwise, it suggests, in its next round, a new value which is the value of the weak shared coin flipped by Procedure COMMUNICATE.

In the code of the algorithm, the variable *round* generates the local order required by the local-order assumption (Assumption 3.1). Each entry of the communication vector is composed of two fields associated with the communication variables: *round* and v. The value of the variable *maxround* is the maximum of all the *round* fields in the communication vector. Procedure COMMUNICATE is either Procedure SM-COMMUNICATE or Procedure MP-COMMUNICATE and procedure HELP-COMMUNICATE is either Procedure SM-HELP-COMMUNICATE or Procedurre MP-HELP-COMMUNICATE.

Note that both Algorithm CONSENSUS and Procedure COIN call Procedures COMMUNICATE and HELP-COMMUNICATE. Furthermore, a processor while flipping one coin may need to help other processors to flip earlier

```
Algorithm CONSENSUS (v, α, cons);
Input    v: an initial binary value;
         α: a bias parameter;
Output   cons: a decision binary value;
var      ε: a binary value;
         round, maxround: counters;
         q: a processor name;
         X: a communication vector;
         (X.round, X.value): the two fields in each entry of X;
begin
  round := maxround := 0;
  for each processor q do X(q) := (0, −1);
  while (round ≠ maxround) or (∃q X(q).value ≠ v and X(q).round ≥ round − 1) do
  (* stop only if round is a maximum and all that disagree trail by at least 2 *)
    round := round + 1;
    X(p) := (round, v);
    COMMUNICATE(X);
    maxround := max_{1≤q≤n}{X(q).round};
    (* if all processors with round = maxround agree on the same value *)
    if ∃ε s.t. ∀_{q,X(q).round=maxround} X(q).value = ε
    then v := ε;  (* adopts the majority value *)
    else v := COIN(round + 1, α);  (* flip a weak shared coin for v *)
  end-while
  (* Here, (round = maxround) and (X(q).value ≠ v iff X(q).round < round − 1) *)
  cons := v;
  HELP-COMMUNICATE(x);
end;
```

Fig. 5. Algorithm CONSENSUS for processor p with initial value v and bias parameter α.

coins. This means that a processor while executing Procedure COMMUNICATE is concurrently executing several instances of Procedure HELP-COMMUNICATE. For simplicity, we omit the mechanism that enables processors to distinguish between different calls and to handle different calls. We just remark that it can be done using the variable *round* of Procedure coin and the variable *round* of Algorithm CONSENSUS.

We omit the proofs of the following lemmas since they are similar to their counterpart proofs in [AH] (Lemma 1 through Theorem 6 and Corollary 14). They are stated so as to emphasize the way the global-order requirement and the deadlock-free requirements are used.

The global-order lemma, Lemma 4.2, implies the following two lemmas.

Lemma 5.5. *In a certain round at most one value can be adopted.*

Lemma 5.6. *If a processor decides on ε in round r, then no other processor suggests $1 - \varepsilon$ in round r.*

The following corollary guarantees the first requirement of the consensus problem (Definition 5.1).

Corollary 5.3. *If a processor decides on ε in round r, then all the other processors decide on ε no later than round $r + 1$.*

The deadlock-free lemma, Lemma 4.3, guarantees the second requirement of the consensus problem (Definition 5.1).

Lemma 5.7. *If all the processors have the same initial value, then they all decide on this value after the second round.*

The expected number of rounds for Algorithm CONSENSUS depends on the type of coin used. If the processors share a perfect (nonbiased) global coin, then the expected number of rounds is constant. On the other hand, if each processor flips a local coin, then the adversary can force the expected number of rounds to be $O(2^n)$. Here, the weak shared coin (Section 5.1) is used, which, by the deadlock-free lemma (Lemma 4.3), is sufficient to guarantee a constant expected number of rounds.

Lemma 5.8. *The expected number of rounds of Algorithms* CONSENSUS *is constant.*

Combining Corollary 5.3, Lemma 5.7, Lemma 5.8, and Theorem 5.1 together yields the following theorem.

Theorem 5.2. *Algorithms* CONSENSUS *using Procedure* COIN, *solves the consensus problem in $O(n^2)$ expected number of rounds.*

6. The Renaming Problem

Definition 6.1. In the renaming problem, each processor p initially has a distinct old name taken from some (possibly large) ordered domain. The goal of the processors is to select a new name from a space of size N, so that every two selected new names are distinct.

The value of N does not depend on the original names, but only on n. For the sake of simplicity, in the description of the algorithm the processors are still identified with the numbers $1, \ldots, n$. However, this numbering is not common to all the processors, otherwise they can solve the problem by selecting these numbers. Nevertheless, the processors are still able to compare entries of their own communication vector and other communication vectors. They compare them by the old name part of the entries. The algorithm guarantees that the communication variables will always contain the old name.

The motivation and the importance of this problem are well explained in the paper [ABD$^+$], which presented two solutions that are based on Algorithm SCHEMATIC. In this section we present the simple algorithm of [ABD$^+$], which solves the renaming problem with a new name-space of size $O(n^2)$.

```
Algorithm RENAMING (ID, NEWID);
Input    ID: the old name of p;
Output   NEWID: the new name of p;
var      S: a set of old names;
         r; integer;
         q: a local name for processors;
         X: vector of n old names;
begin
  for q := 1 to n do X(q) := 0;
  X(p) := ID;
  COMMUNICATE(X);
  S := the set of the old names that appear in X;
  r := the rank of ID in S;
  NEWID := (|S| choose 2) + r;
  HELP-COMMUNICATE(X);
end;
```

Fig. 6. Algorithm RENAMING for processor p.

The goal of every processor is to learn the names of other processors. Due to the asynchronous fail-stop environment, it cannot do it. The idea behind Algorithm RENAMING is that processors select their new names according to some sets of old names only if any two such sets are comparable (one set is a subset of the other set). This condition can be achieved by one call to Procedure COMMUNICATE (either Procedure SM-COMMUNICATE or Procedure MP-COMMUNICATE).

Algorithm RENAMING appears in Figure 6. The communication variables of a processor in this solution contain only the old name of this processor. Here processors compare two entries of two vectors by their content (old names) and not by their index. If a processor does not have an entry containing some old name and it sees this old name in another vector, it deduces that it has less updated information. Note that there are two possible values to each entry: either the null value or an old name of some processor.

Again, since processors cannot quit before all the other processors make their decision, they call Procedure HELP-COMMUNICATE after selecting their new name.

Denote by S_k the set of the old names that appear in the kth stable vector X_k. The following are two simple corollaries of the gobal-order lemma (Lemma 4.2).

Corollary 6.1. *If* $1 \leq i \leq j$, *then* $S_i \subseteq S_j$.

Corollary 6.2. *For every* l, $1 \leq l \leq n$, *if* $|S_i| = |S_j| = l$, *then* $S_i = S_j$.

In the algorithm, each processor p invokes Procedure COMMUNICATE only once and then decides according to the output stable vector X. Let l be the number

of old names that appear in X, and let r be the rank of p's old name along these old *ID*s. Then processor p selects $\binom{|S|}{2} + r$ as its new name.

The following two lemmas prove that the algorithm is correct and that all nonfaulty processors select a new name. Namely, each nonfaulty processor selects a distinct new name after a finite number of operations.

Lemma 6.1 (Correctness). *In any run of Algorithm* RENAMING, *if p and q select the new names x_p and x_q, respectively, then $x_p \neq y_q$.*

Proof. Let

$$x_p = \binom{s_p}{2} + r_p$$

and let

$$x_q = \binom{s_q}{2} + r_q.$$

If $s_p = s_q$, then, by Corollary 6.2, p and q are decided according to the same set of old names. Therefore, their ranks, r_p and r_q, are not equal and consequently $x_p \neq x_q$. If $s_p \neq s_q$, then assume without loss of generality that $s_p < s_q$. As $r_p \leq s_p$ it follows that

$$\binom{s_p}{2} + r_p < \binom{s_q}{2} + 1.$$

Since $1 \leq r_q$ it follows that

$$\binom{s_p}{2} + r_p < \binom{s_q}{2} + r_q$$

and therefore $x_p \neq x_q$. □

Lemma 6.2 (Termination). *All the nonfaulty processors select a new name.*

Proof. The deadlock-free lemma (Lemma 4.3) guarantees that at least one non-faulty processor selects a new name. However, this processor makes his decision immediately after outputting the first stable vector. The claim of this lemma is implied by applying again and again the deadlock-free lemma. □

By Lemmas 6.1 and 6.2 and by the fact that the number of possible new *ID*s is exactly

$$\binom{n+1}{2} = O(n^2)$$

we get the next theorem.

Theorem 6.1. *Algorithm* RENAMING *solves the renaming problem with a new ID space of size* $\binom{n+1}{2}$. *This solution is valid in both the shared-memory model and the message-passing model defined in Section* 2.

In the message-passing model the size of the new space is of order $O(nt)$ [ABD$^+$]. The second algorithm of [ABD$^+$] solves the renaming problem with a new name-space of size $2n - 1$ ($n + t$ in the message-passing model). Since it is also based on the schematic algorithm, Theorem 6.1 can be improved accordingly.

A related problem to the renaming problem requires the following: if the old name of processor p is smaller than that of processor q, then the new name of p is smaller than the new *ID* of q. In the paper [ABD$^+$] this problem is also solved. Again their solution is based on the schematic algorithm and, therefore, is valid in both the shared-memory model and the message-passing model defined in Section 2.

Acknowledgment

We would like to thank the anonymous referees whose comments helped us to improve the presentation of this paper substantially.

References

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

[AH] J. Aspnes and M. Herlihy, Fast Randomized Consensus Using Shared Memory, *Journal of Algorithms*, **11**, 441–461, 1990.

[ADB$^+$] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk, Renaming in an Asynchronous Environment, *Journal of the ACM*, **37**, 524–548, 1990.

[ADS] H. Attiya, D. Dolev, and N. Shavit. Bounded Polynomial Randomized Consensus, *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pp. 281–293, 1989.

[B] M. Ben-Or, Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols, *Proc. 2nd ACM Symp. on Principles of Distributed Computing*, pp. 27–30, 1983.

[BMZ] O. Biran, S. Moran, and S. Zaks, A Combinatorial Characterization of the Distributed Tasks Which are Solvable in the Presence of One Faulty Processor, *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pp. 263–275, 1988.

[BW] M. F. Bridgland and R. J. Watro, Fault-Tolerant Decision Making in totally Asynchronous Distributed Systems, *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pp. 52–63, 1987.

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

[CMS] B. Chor, M. Merritt, and D. Shmoys, Simple Constant-Time Consensus Protocols in Realistic Failure Models, *Journal of the ACM*, **36**, 591–614, 1989.

[CM] B. Chor and L. Moscovici, Solvability in Asynchronous Environments, *Proc. 30th Symp. on Foundations of Computer Science*, pp. 422–427, 1989.

[DDS] D. Dolev, C. Dwork, and L. Stockmeyer, On the Minimal Synchronism Needed for Distributed Consensus, *Journal of the ACM*, **34**, 77–97, 1987.

[DS] D. Dolev and N. Shavit, Bounded Concurrent Time-Stamp Systems are Constructible, *Proc. 21st ACM SIGACT Symp. on Theory of Computing*, 1989.

[DLS] C. Dwork, N. Lynch, and L. Stockmeyer, Consensus in the Presence of Partial Synchrony, *Journal of the ACM*, **35**, 288–323, 1988.

[DSS] C. Dwork, D. Shmoys, and L. Stockmeyer, Flipping Persuasively in Constant Expected Time, *Proc. 27th Symp. on Foundations of Computer Science*, pp. 222–232, 1986.

[Fe] P. Feldman, Private communication.

[Fi] M. J. Fischer, The Consensus Problem in Unreliable Distributed Systems (a Brief Survey), YALEU/DCS/RR-273, June 1983.

[FLP] M. J. Fischer, N. A. Lynch, and M. S. Paterson, Impossibility of Distributed Consensus with One Faulty Processor, *Journal of the ACM*, **32**, 374–382, 1985.

[GHS] R. G. Gallagher, P. A. Humblet, and P. M. Spira, A Distributed Algorithm for Minimum-Weight Spanning Trees, *ACM Transactions on Programming Languages and Systems*, **5**, 66–77, 1983.

[H] M. P. Herlihy, Impossibility and Universality Results for Wait-Free Synchronization, *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pp. 276–290, 1988.

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

[L] L. Lamport, On Interprocess Communication, Part I and II, *Distributed Computing*, **1**, 77–101, 1986.

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

[PSL] M. Pease, R. Shostak, and L. Lamport, Reaching Agreement in the Presence of Faults, *Journal of the ACM*, **27**, 228–234, 1980.

Received January 10, 1990, and in revised form October 10, 1990, and November 11, 1990, and in final form December 20, 1991.