On Processor Coordination Using Asynchronous Hardware

Benny Chor*
Aiken Computation Lab.
Harvard University

Amos Israeli†
Aiken Computation Lab.
Harvard University

Ming Li‡
Harvard University
and Ohio State University

Abstract

We investigate an asynchronous model of concurrent computations, where processors communicate by shared registers that allow atomic read and write operations (but do not support atomic test-and-set). For this model, we define a general notion of processor coordination, and study the possibility and complexity of achieving coordination. Our definition includes, as special cases, mutual exclusion and asynchronous agreement. It is shown that the coordination problem cannot be solved by means of a *deterministic* protocol even if the system consists of only two processors. This impossibility result holds for the most powerful type of shared atomic registers and does not assume symmetric protocols. The impossibility result is contrasted by a variety of efficient randomized protocols, that achieve fast coordination for systems of arbitrary number of processors n. These protocols are all fairly simple, constructive, and their *expected* run-time is polynomial in n, even in the presence of an adaptive adversary scheduler. All our protocols use only the most restricted type of registers in this class, namely single reader, single writer, bounded size registers. These registers and hence our system are implementable in existing technology.

*Supported in part by NSF Grant MCS81-21431 at Harvard University.

†Supported in part by The Weizmann fellowship and NSF Grant DCR-86-00379.

‡Supported in part by ONR Grant N00014-85-k-0445 at Harvard and by NSF Grant DCR-86-06366 at OSU.

1 Introduction

The problem of coordinating the actions of different processors in a distributed environment is one of the most fundamental problems whenever any type of cooperation is to be achieved. The nature of solutions to this problem clearly depends on the properties of communication media and on the relative speeds of participating processors. In this paper we investigate the coordination problem under no assumptions on processor speeds or communication delays. In other words, we are dealing with a totally asynchronous system.

The coordination problem we study is the following: Every processor starts the protocol with an arbitrary input value (for example, an externally supplied variable or an internally computed constant). Upon termination, each processor irrevocably decides on an output value. We have two requirements from the output. The first is that all processors which have terminated hold the same output value. The second is that the output value must be one of the input values of active processors.

This coordination problem is very general, and includes several well studied distributed problems as a special case. For example, the mutual exclusion problem can be formulated in our con-

text as choosing the identity of a processor who is to enter the critical region. In this case, the input value of every processor in the trial region is simply its own identity.

The communication mechanism we consider are shared registers that are atomic (serializable) with respect to *read* and *write* operations. It is convenient to think about all the *read* and *write* operations in terms of a global time model. In this model each such I/O operation takes place in a closed interval on the global-time axis. Atomicity of a register means that every set of *reads* and *writes* from/to this register is equivalent to a sequence in which each operation is shrunk to a different point inside its corresponding time interval, hence all these operations are totally ordered. For a precise definition of atomicity, we refer the reader to the important paper of Lamport [5], where this notion is discussed extensively. The notion of atomic *read* and *write* is much less restrictive than another type of atomic operation that is sometimes used in the literature, namely atomic *test-and-set*. In fact, atomic test-and-set seems to require quite stringent timing constraints on the low level hardware – that read, test, and write cycles of different processors do not overlap in time.

Our model is especially appealing because it can actually be implemented. All communication is done through bounded size single writer single reader registers. As proved in [5], these registers can be implemented from existing low level hardware. This model looks more realistic than those using unbounded message buffers which are delt with in some other works. At the expense of somewhat weakening the adversary by not allowing byzantine behaviour, we can handle arbitrary speed differences between the participating processors. Actually we account to fail/stop type errors of up to all but one of the system processors.

It is not hard to see that the use of atomic registers allows us to serialize not only the actions corresponding to a single register, but an entire system execution. A serialization should specify a total order on all operations in such system execution. Actions of the same processor are ordered according to their internal order. Actions of several processors which share the same register are ordered by the atomic register. Actions of several processors with different registers can be serialized in every possible way which is compatible with the last two rules. No inconsistencies will result, since all these possible serializations lead to equivalent executions. Therefore from now on we assume that in every system execution all actions occur in separate time units. Under this assumption every system execution can be associated with a *schedule*, which is an ordered list of processors of the system, corresponding to the order of their actions. However, any system execution with overlapping operations can be serialized in many different ways, and we do not know apriori which order of execution will be chosen. In reality, this order may depend on complicated physical phenomena, perhaps at the quantum level, which we do not know or wish to analyze. We therefore consider the worst possible sequencing of events, by viewing the scheduler as an adversary.

Any solution to the coordination problem should satisfy the *consistency* and *termination* requirements. Consistency means that no execution leads to different decision values by any two processors. Termination means that every processor which is activated a sufficient number of times will decide and terminate. Of course, we cannot require that slow processors that hardly take any steps will terminate. We'd like to have a solution which guarantees that every schedule in which a processor is activated at least k times (for some constant k) leads to termination by that processor. This implies, in particular, that no processor should wait for other processors to take steps – it should terminate regardless of whether or not other processors were active in between its own steps (of course, the output value could depend on other processors' activity). Such requirement is in accordance with the complete asynchrony of the system: It does not make sense to force the very fast processors to wait until a very slow processor makes a move. In particular, our termination condition implies that for any "fair" schedule (where every processor is activated infinitely many times), all processors should terminate.

Unfortunately, our goal as stated above cannot be achieved. Our first result is an impossibility result that holds for any number of processors $n \geq 2$. We show that if the processors in a system of size n are using *deterministic* protocols, then the coordination problem cannot[1] be solved. In other words, there is an initial legal configuration of the system and an *infinite* schedule, under which no processor ever terminates. (This result holds without any further restriction on the nature of the deterministic protocols used by the processors. In particular, they are not required to be symmetric.) This impossibility proof owes a lot to the impossibility proof of Fischer, Lynch, and Paterson [4]. The conceptual structure of the two proofs is the same. Despite this similarity, these two results are not comparable. In particular an impossibility result in the message passing model does not necessarily imply such a result in our model. This is best demonstrated by the fact that in the message passing model of [4] no agreement (even randomized) can be achieved if more then half of the processors are faulty [2]. Our protocols, on the other hand, reach such agreement even in the case of $t = n-1$ possible crashes among n processors.

It is by now a well known fact in the area of distributed computing that certain problems which cannot be solved by deterministic protocols do admit randomized solutions [6], [7], [1]. It is then only natural that the next step we take is to overcome the abovementioned impossibility result by allowing processors to toss coins. We present efficient randomized protocols, that achieve fast coordination for systems of arbitrary size n. These protocols are all fairly simple, constructive, and their *expected* run-time is polynomial in n. (The probability that a processor does not terminate after taking kn steps is bounded above by an exponentially decreasing function of k. When considering a schedule in a randomized setting, one has to specify the "dynamic" knowledge of the scheduler. It might be advantageous to an adversary scheduler to know the internal states of individual processors before assigning the next processor to take a move. Indeed, We allow the scheduler to have complete knowledge on both registers' contents and processors' internal states. Even with this adaptive adversary, the protocols *never* err – no execution leads to contradicting outputs. Therefore they achieve our previously stated goal, even in the worst possible environment, if we add the clause "with very high probability". Every processor that makes sufficiently many moves will terminate with very high probability.

Our first protocol is a randomized coordination protocol for a system of two processors. In this protocol, every processor terminates after an expected number of 10 steps. The protocol is very simple, and requires only one bit shared register per processor. We then move on to larger size systems. It turns out that the simple protocol of $n = 2$ cannot be directly extended. We develop different randomized protocols that work for any n. The protocols are still fairly simple, and as mentioned above, have polynomial (in n) expected run-time. The first protocol in this class uses unbounded size registers (though large values are actually written only with very low probability). The main usage of the unboundedness is is to maintain a *global* ordering of processors' actions, which is known to the processors themselves. Such property cannot be satisfied with bounded size registers. Surprisingly, we managed to modify the protocol to one which uses only *bounded size* registers. This is done by maintaining *local* ordering of processors' actions. This (non-transitive) "ordering" turns out to be sufficient for solving the coordination problem.

The remainder of this paper is organized as follows: In Section 2 we formally define the model, the class of admissible schedules, and the coordination problem. In Section 3 we prove the impossibility of deterministic coordination. In Section 4 we present the protocol for a system

[1] Our impossibility result for deterministic protocols *does not* contradict the existence of deterministic mutual exclusion algorithms a-la Dijxtra [3]. The reason is that these algorithms are correct only with respect to a partial set of schedules, namely, *admissible schedules*. In these schedules, any processor that is activated only finitely often eventually reaches a special area in its program called the remainder section. Thus, schedules where, for example, a processor is held out sometime before entering its critical region, could yields a deadlock. Such a schedule is not admissible but can be handled in our context.

with two processors. In Section 5 we present the protocol for a system with 3 processors and unbounded shared registers. In Section 6 we present the bounded registers three processor protocol. In the full paper we will generalize these last two protocols to n processor protocols.

2 Model and Definitions

In this section we define our model of asynchronous distributed computation, the coordination problem, and the class of schedulers we are interested in. This section is (necessarily) quite formal. Many of our definitions draw upon those in [4], with extentions to handle probabilistic choices. We deviate from [4] in two major ways: The first difference is that the requirements from the protocol are for performance with respect to *finite* schedules. the second difference between the models lies in the communication media. The communication medium we are studying is more restrictive than the message passing mechanism under which the asynchronous agreement was traditionally studied. The message buffers in these systems were assumed to have the capability of holding unlimited number of *different* messages, whereas a shared register can hold only one message at a time.

An asynchronous distributed system is a collection of n processors. Every processor P is a (not necessarily finite) state automaton with an internal input register i_P and an internal output register o_P. The value in the input register is from a set V, while the output register has initially the value $\perp$ ($\perp \notin V$) and could be changed once to a value in V. The set of all states of processor P will be denoted by S_P. The set S_P contains a distinguished state I_P which is the *initial state* of the processor P. States in S_P where o_P contains a value $\neq \perp$ are called the *decision states* of processor P. The set S_P might be infinite thus every internal state can code the whole history of the computation of the processor P.

Processors communicate via *shared registers*. Every shared register r is associated with a set of processors R_r, $\mid R_r \mid > 0$, that can read the register, and a set of processors W_r, $\mid W_r \mid > 0$, that can write into the register. (To eliminate redundant cases we assume $R_r \neq W_r$) These registers are atomic with respect to the read and write operations.

Every processor P takes steps according to its *transition function* T_P. Each step consists of a single input/output operation, followed by a state transition. The transition function T_P determines either a register for a read operation, or a value and a register for a write operation. In case the communication action is a read, the new state of P depends on the value read by this action. The transition function T_P could be either deterministic or probabilistic. In the latter case, for every state $s \in S_P$, there is a probability measure assigned to the next step. Given an asynchronous system as specified above, a *protocol* is a collection of n transition functions $T_1, \ldots, T_n$, one per processor.

A *configuration* C of the system consists of the state of each processor together with the contents of the shared registers. In an *initial configuration*, every processor is in its initial state, and all shared registers and output registers, contain the default value $\perp$. The set of all configurations will be denoted by $\mathcal{C}$. A *step* takes one configuration to another by activating a single processor P. That is, a step is a single application of one transition function T_P. A *run* of length ℓ is a sequence of ℓ steps. Each run has a *schedule* which is a sequence of ℓ processor numbers according to the order of processors that take steps in that run. We denote schedules, finite or infinite, by a list of processor numbers, e.g. $(2,3,3,2,1)$. If S is a finite schedule then we denote by $S \circ i$, where i is any processor number, the schedule obtained from the schedule S by concatenating the number i to the end of S. We say that processor P is activated k times in a run if P appears k times in its schedule.

A *scheduler* S is a mapping from $\mathcal{C}$ into the set of n processors. Given the configuration of the system, the scheduler picks the next processor that is to take a step. The scheduler could either be a deterministic mapping or a probabilistic one. The scheduler is best viewed as an adversary that tries to prevent us from reaching

our goal. Under the definition, this adversary is the strongest one possible: It has complete knowledge on the state of every processor and on the contents of the shared registers. (Notice that our formalization of processors easily allow for the inclusion of the complete history of a run as part of the state of processors, and thus the scheduler need not explicitly have the history as an argument for determining the next activated processor.) In case the processors are probabilistic, the scheduler could also base its choices on the outcome of past coin flips. We do not allow it, though, to be able to predict *future* probabilistic moves of the processors. This is a necessary requirement if randomization is to be helpful at all, and it is used in all algorithms where randomization is employed *e.g.* [6], [7], [1]. Given a configuration C and a scheduler S, the runs which can be produced by S, starting from C, on some possible probabilistic choices are called the runs *compatible* with a C and S. Notice that if both processors and scheduler are deterministic, then there is a single compatible run starting from every C.

We say that a configuration C_2 is reachable from configuration C_1 with schedule S if there is a run compatible with C_1 and S which leads to configuration C_2. We say that a configuration has a decision value v if some processor P is in a decision state with its output register o_P containing $v \neq \perp$.

A *coordination protocol* is designed for an asynchronous system of $n \geq 2$ processors. We classify a protocol as either deterministic or randomized, according to the nature of the transition functions. The protocol specifies a set V of possible inputs whose cardinality is at least two (otherwise the problem is trivial), and a default value $\perp$ that is not a member of V. It is required to satisfy the following properties:

1. **Consistency:** for every schedule, no configuration reachable from an initial configuration has more than one deciding value.
2. **Nontriviality**: if processor P has decided on value v in a run, then v is an input value for at least one processor which was activated in the run. (That is, the output value is an input of at least a single active processor.)
3. **Termination**
   - **Deterministic Termination**: there is a constant k so that for every initial configuration C_0, for every schedule, if a processor P was activated more than k times, then P is in a deciding state.
   - **Randomized Termination**: there is a function f from the natural numbers into the interval $[0,1]$ satisfying $\lim_{k\to\infty} f(k) = 0$, so that for every initial configuration C_0, for every schedule, if a processor P was activated more than k times, then with probability $\geq 1 - f(k)$, P is in a deciding state.

It should be noticed that we require that randomized coordination protocols will never err. We only allow longer runs, but these should occur with vanishing probabilities for longer and longer runs.

3 Impossibility of Deterministic Coordination

In this section we prove that there is no deterministic protocol which solves the coordination problem even for the very restricted system with only two processors and two decision values. We show that every deterministic protocol which guaranties consistency and nontriviality, has an *infinite* schedule under which no processor ever terminates.

Let ST be a deterministic system with two processors P_1 and P_2 and two decision values a and b. Two runs of a system are *equivalent* for a processor P_i if this processor takes the same steps in both runs. If a processor starts two runs with the same input value, and if it reads the same values in all its *read* actions than the two runs are equivalent for it. If a processor P_i reaches a decision state in a run it reaches the same decision state in every run of the system which is equivalent for P_i. Note: Two runs which are equivalent for one processor do not necessarily have even a

single equal configuration. Two runs are *equivalent* if they are equivalent for all processors. A configuration C of ST is called *bivalent* if there exist two schedules S_1 and S_2 and two decision configurations C_1 and C_2 with decision values a and b respectively such that C_1 is reachable from C using S_1, and C_2 is also reachable from C by using S_2. A configuration of S is *univalent* if it is not bivalent. A univalent configuration has a single decision value. We first prove an easy technical lemma.

Lemma 1: Let C be a bivalent configuration of of a coordination protocol CP, for ST. Then C is not a decision configuration.

Proof: Assume that C is a decision configuration. Then one processor, say P_1 which is in its decision state with a certain dcision value, say v_1. Since C is bivalent there is a second decision value, say v_2 and a schedule S such that activating S from C will result in the decision of v_2. But since the value in o_1, the output register of P_1 cannot be changed we get a configuration with different decision values in different output registers, in contradiction with the consistency of CP. □

We are now ready to prove the inexistence of a coordination protocol for ST. We will do that by showing that for every consistent and nontrivial deterministic coordination protocol we can construct an infinite schedule under which no processor will ever terminate. This schedule will be constructed inductively. We will first show that every nontrivial protocol has a bivalent initial configuration.

Lemma 2: Let CP be a coordination protocol for ST. Then CP has a bivalent initial configuration.

Proof: Consider the initial configuration I_{aa} where the input value for both P_1 and P_2 is a. By the the nontriviality property for for Coordination Protocols the output value of CP must be the value a regardless of the schedule. In particular the decision value for the schedule $S_1 = (1, 1, 1, \ldots)$ is also a. Note that in this schedule the processor P_1 never reads any value written by P_2. Analogously the decision value reached by CP on I_{bb} must be the value b. Consider the initial configuration I_{ab} where the input P_1 is a and the input value for P_2 is b. The decision value reached by P_1 on S_1 must be a since the processor P_1 cannot distinguish between this run and the run with I_{aa} with S_1. In the same way the decision value for P_2 with I_{ab} and $S_2 = (2, 2, 2, \ldots)$ must be b. Thus I_{ab} is a bivalent configuration. □

We proceed by showing that each bivalent configuration of ST, has a reachable configuration which is also bivalent.

Lemma 3: Let C be a bivalent configuration of ST with CP such that applying P_1 to C leads to a univalent configuration C_1 with decision value a. Then the configuration C_2, the result of applying a step of P_2 to the configuration C, is bivalent.

Proof: Assume, towards a contradiction, that the configuration C_2 is also univalent. In this case the decision value of C_2 is b, by the bivalence of the configuration C. By the termination property of CP the schedule $S_{22} = (2, 2, 2, \ldots)$ leads to a decision state whose decision value is b.

The action performed by the processor P_1 from the configuration C must be a *write* action, otherwise the schedules $S_{12} = (1, 2, 2, 2, \ldots)$ and $S_{22} = (2, 2, 2, \ldots)$ applied to the configuration C produce an equivalent run for the processor P_2. Hence these two runs result the same decision value, namely the value b, in contradiction to the univalence of C with S_1. For the same reason the action performed by P_2 on S_2 from C must also be a *write* action.

These two *write* actions should be to the same register for if they were not then the configuration C_{12} reached by applying the schedule $(1, 2)$ to the configuration C is the same as the configuration C_{21} reached by applying the schedule $(2, 1)$ to C. But this is impossible since both are univalent configurations with different decision values.

Finally, assume that those actions are indeed *write* actions to the same register. In this case

the decision value of the schedule $(2,1,1,1,\ldots)$ applied to the configuration C is the same as the decision value for the schedule $(1,1,1,\ldots)$, namely the value a. This is due to the fact that the *write* action of P_1 erases the value written by P_2. Thus those last two runs are equivalent for P_1 and thus must have the same decision value.

This forms the final contradiction which is caused by the assumption that the configuration C_2 is univalent. Therefore this configuration is bivalent. □

Theorem 4: There is no Coordination Protocol for ST.

Proof: Assume there was such a protocol CP. We show that CP has an infinite run, therefore it contradicts the termination property for coordination protocols. We construct now inductively an infinite run whose all configurations are bivalent. By lemma 2 the initial configuration of CP, I_{ab}, must be bivalent. Assume now that we have constructed a schedule S_l of length l such that all configurations reached by applying S_l to I_{ab} are bivalent. In particular the configuration C_l reached after these l steps of S_l is bivalent. By lemma 3 either $S_l \circ 1$ or $S_l \circ 2$ leads to a bivalent configuration, C_{l+1}. □

4 The Two Processor Protocol

The two processor protocol is a very simple and lucid coordination protocol, which dramatically demonstrates the added power of randomization. Before presenting the two processors protocol, we give a lemma stating that in order to solve the coordination protocol for any k value set, it suffices to solve the coordination problem for a binary value set.

Theorem 5: Let CP_2 be a coordination protocol for a system with n processors with two decision values. A coordination protocol CP_k for n processors with an arbitrary number k of decision values can be constructed using CP_2. The complexity of CP_k is $\log k$ times larger then he complexity of CP_2.

The intuitive idea of the two processor protocol with binary decision values a, b is the following: At any time each activated processor has a preferred decision value. The processor repeats reading the value preferred by the other processor. If these two preferred values are the same, value then the processor decides on this value and quits. If however different values are preferred, then the processor flips an unbiased coin, either changes its preferred or rewrites the old preferred value [2], and reads again. For any given configuration, if it is not univalent already, then with probability at least $1/4$, one of the two next write steps will lead to a univalent configuration. The adversary scheduler, not knowing what value the processor will write next, cannot prevent reaching a univalent configuration with probability at least $1/4$, regardless of what two processors take the next two write steps.

We describe the protocol for two processors P_0 and P_1. Each processor has a register, r_0 and r_1 respectively in which it has its current preferred decision value. Processor P_0 can write into r_0 and read from r_1. The last value read from r_1 is stored in an internal variable v_0. These two shared registers are initialized to a default value, $\perp$. The protocol for P_0 is shown in Figure 1.

```
    { initially r0 = v0 = ⊥}
(0) write r0 ← input.
    repeat
(1)         read v0 ← r1.
            if v0 = r0 or v0 = ⊥
            then decide r0 and quit.
(2)         else
                    Flip an unbiased coin.
                    if Heads
                    then rewrite r0 ← r0,
                    else  write r0 ← v0.
    until decision is made.
```

Figure 1: The two processor protocol (for P_0):

[2] This rewriting action is actually superfluous and is used only for ease of analysis

To show correctness of this protocol we should prove consistency, non-triviality, and (randomized) termination. Non-triviality is trivial for this protocol, so we prove the other two properties.

Theorem 6: The protocol is consistent.

Proof: We show that if any processor chooses a decision value v, and the other processor later choses a decision value, these two values will be the same. Suppose, without loss of generality, that P_0 was the first to choose and that the value chosen is v. At the moment when P_0 read the value v in r_1 the values of r_0 and r_1 are both v. Since P_1 has not chosen a value yet, he must read r_0 at least once before making its choice. But the value of r_0 will not change afterwards, so it is impossible that P_1 will read a different value in r_1 later. hence P_1 will also choose v. □

We prove randomized termination even for *adaptive schedulers* S who know the internal state of both processors.

Theorem 7: The protocol satisfies the randomized termination condition.

Proof: Let S be an arbitrary adaptive scheduler, By analyzing all possible bivalent configurations (there are not too many of them), we will show that with probability at least $1/4$, after two write steps are made, the system will be in a univalent configuration. The possible configurations are completely characterized by the contents of the two shared registers, the two internal variables, and the program counter of both processors. We say that processor P_i is *about to read* if its program counter is at 1. We say that a processor is *about to write* if its program counter is at 2. (Notice that the other instructions are the internal computations associated with the read and write operations, respectively.)

The basic fact we use is that if a processor is about to write then with probability $1/2$, $r_0 = r_1$ will hold after its write operation. Suppose that both processors are about to write. Without loss of generality, S activates P_0 first. With probability $1/2$, the value that P_0 writes will cause $r_0 = r_1$. Suppose this has happened. Now P_0 is about to read. If it reads in the next step, it will decide, and our claim is justified. If in the next step P_1 writes, then with probability $1/2$ he will not change r_1. Now they both are about to read, and whoever is activated first will decide. The other configurations are analyzed similarly.

We conclude that with probability at least $1/4$, every pair of read-write operation (except the initial write) by the processor P_i will lead to a decision by P_i. Taking into account the initial write operation and the "final" read operation, we see that after $k+2$ steps of P_i, the probability that P_i has not decided yet is $\leq (1/4)^{k/2}$. Thus termination holds with an exponentialy deacreasing function $f(k)$. □

Corollary: The expected number of steps by P_i to decide is $\leq 2 + 4 \cdot 2 = 10$.

5 Coordination Protocol for Three Processors

In this section we present a very simple and efficient randomized coordination protocol for three processors using unbounded registers. There is a positive (though very small) probability for very large numbers to be written in the registers. In the Section 6 we present a (much more complicated) bounded register protocol.

When designing a coordination protocol for more than two processors one must be very carful. Many "natural" protocols fail in very subtle ways which are far from obvious at first site. Consider for instance the following "Consensus Protocol". Each processor chooses at random a value, out of a and b. When all processors have chosen the same value they terminate. This simple and appealing protocol failes because the adversery has the following strategy: first activate P_1 once and choose the value a with probability $1/2$. Then activate P_2 and choose the value b with probability $1/2$. In this configuration activate P_3 for ever. No decision can be reached by P_3.

The following protocol overcomes such difficulties and still maintains conceptual simplicity. Each processor has a 1-writer 2-reader commu-

nication register. In the full paper we prove that the same protocol also works with 1-writer 1-reader registers. Each processor maintains in its register a *pref* field and a *num* field. The possible values of the *pref* field are *bot*, *a* and *b*. The possible values of the *num* field are the non-negative integeres. All communication registers are initialized to $\perp$ in the *pref* field and 0 in the *num* field.

Intuitively the protocol works like this. Each processor keeps its currently preferred decision value in its *pref* field. The *num* field is used to keep the ordering of the processors. After reading the registers of the other two processors, the processor computes the new contents of its own register. Let the *leading* processor(s) be the processor(s) whose *num* is maximal. A decision is made in two cases. The first case is when the *pref* of all processors is the same. The second case where a decision is made is when the *pref* of all the leading processors is the same and their *num* is greater by two or more from the *num* of the other processors. In this case the leading processor's *pref* is the decision value. If none of these conditiond hold then a new register contents should be computed. In order to break symmetry this new contents is only used in half of the time. In the other half, the old register contents is retained. The new *pref* is computed as follows: In case all leading processors have the same *pref* this value is taken by the other processors. If the leading processors have two different *pref*s then each processor keeps its own *pref*. The new *num* value is the old *num*+1. The protocol appears in Figure 2.

Theorem 8: The protocol is consistent.

Theorem 9: In this protocol, the probability of *num.* $= k$ in any register is at most $(3/4)^k$.

Proof: We consider only the worst case in which the *num* fields of all registers always differ by at most 1. Because each time when a processor increases its *num* field by 1 and becomes the leading processor the other 2 processors has 1/4 probability of agreeing with this leading processor, therefore with at most probability 3/4 they go into another round. The theorem follows directly from this. □

Corollary: The expected running time of the protocol is a small constant.

6 Three Processors with Bounded Registers

We describe an algorithm for three processors using bounded registers. Here we give a protocol which uses 1-writer 2-reader registers. In the full paper we modify the protocol for 1-writer 1-reader registers.

Let the three processors be P_1, P_2, P_3, each with register r_1, r_2, r_3, respectively. Each register consists of 2 fields [number-field,-field] which can hold the following values:
[1,a], ... , [9,a],
[3,*pref*-a], [6,*pref*-a],[9,*pref*-a], [*dec*-a]
[1,b], ... , [9,b],
[3,*pref*-b], [6,*pref*-b],[9,*pref*-b], [*dec*-b]
(As a matter of fact, each register has a third field which will be specified later for simplicity reasons). These values are (circularly) "ordered" as [1,–]<[2,–]< ... <[9,–]<[1,–]< ... where "–" means "don't care". This relation induces a partial ordering on every subset of the registers values which does not include elements of the form [*i*,–] for some *i*. These induced orderings are, of course, not compatible.

Denote the two processor and the unbounded three processor protocols as $\mathcal{A}_2$ and $\mathcal{A}_3$ respectively. If we apply $\mathcal{A}_3$ directly using the above assumptions, a contradiction can be immediately constructed because the circular loop in the register values can confuse the processors. To overcome this difficulty, we introduce a new ingredient into our protocol: At any fixed time, the values of the registers of the three processors are either all in one of the intervals ([8,–],[3,–]), ([2,–],[6,–]), or ([5,–],[9,–]). They always increase from [8,–] to [3,–] (through [9,–],[1,–]), from [2,–] to [6,–], and from [5,–] to [9,–], step by step. Therefore when $r_1 = [m,–]$, $r_2 = [n,–]$, $r_3 = [k,–]$, we can define a "leading" processor to be a processor whose register has $[\max(m,n,k),–]$ (Note: $9 < 1$). Notice that there may be more than

```
newreg.pref← input;
newreg.num← 1;
write newreg;
repeat
        oldreg ← newreg;
        reg2 ← read r2;
        reg3 ← read r3;
        maxnum ← max (newreg.num, reg2.num, reg3.num);
        if pref of all registers is the same
          or
          pref of all leading processors is equal and
          num of all other processors is < maxnum − 1
        then
             decide on pref of leading processor(s);
        else
             toss a fair coin
             if heads
             then
                  if all leading processors have the same pref
                  then
                       newreg.pref ←pref of leading processor;
                  else
                       newreg.pref ← oldreg.pref;
                  endif
                  newreg.num ← oldreg.num+1;
             else (if tailes)
                  newreg ←oldreg; (retain old value)
             endif
        endif
        write  newreg;
until decision is made;
```

Figure 2: The three processor unbounded protocol (for P_1):

one leading processors. We similarly define "second", and "last" processors, and the relation of "l steps behind (apart, away)".

The idea is as follows. P_1, P_2, and P_3, begin with [1,–], start to run the protocol $\mathcal{A}_3$ pretending they have unbounded registers. When a (leading) processor reaches [3,–], or [6,–], or [9,–], it checks if the last processor is at least 2 steps away. If not, then all three processors continue to run $\mathcal{A}_3$ where [1,–] is considered as the successor of [9,–]. If the last processor is at least 2 steps away, then the 2 leading processors (note that they are at most 1 step apart since otherwise they reach agreement by algorithm $\mathcal{A}_3$) start to run the 2 processor algorithm $\mathcal{A}_2$, in order to wait for the 3rd processor to catch up. In the process of $\mathcal{A}_2$, either the 3rd processor catches up or the leading two agree on a value. In the latter case, a decision can be made; In the former case, when the 3rd processor catches up to within 1 step apart to the leading processor, all three processors switch back to $\mathcal{A}_3$.

The protocol, like $\mathcal{A}_3$ works in phases. In each phase a processor reads the registers of the other two processors and then computes its new register which again is written with probability $1/2$. In the rest of the times the old value is retained. The read operation here, however is more complicated; for reasons which are made clear in the proof the protocol works only if the value of the processor ahead is read last. Hence each phase starts with the following sequence: $\mathbf{read}(r_2)$, $\mathbf{read}(r_3)$, and if P_2 is ahead of P_3 then $\mathbf{re-read}(r_2)$.

A detailed description of the protocol, using a transition function diagram, is given below. In this description we use the following facts:

1. Since all processors have the same algorithm, we describe the protocol for P_1. The protocol for P_2 and P_3 can be obtained by exchange P_1 (r_1) with P_2 (r_2) or P_3 (r_3).

2. Since in the three regions ([8,–],[3,–]), ([2,–],[6,–]), and ([5,–],[9,–]) the transition function is the same (modulo 3), we describe the transition function only for the case where all processors are in the region ([8,–],[3,–]) but not all in [2,–],[3,–]. (in this latter case they are considered to be in the region ([2,–],[6,–])).

3. Within region ([8,–],[3,–]), the transition functions from a state [m,a] is the complement of the transition functions from a state [m,b]. We therefore describe only the transition functions from [8,a], ... , [3,a]. The transition functions for [m,b] can be obtained by exchanging "a" and "b".

In Figure 3 we show the state transition diagram for states [8,a], ..., [3,a] of P_1 when *all* the processors are in [8,a]–[3,a] but *not all* in [2,a]–[3,a] which belongs to the definition of ([2,–],[6,–]). In this diagram ovals stand for states. The value of the register r_1 for each state is written inside its oval. On each phase the processor moves with probability 1/2 along one of the exiting arrows. The arrow along which the processor moves depends on the conditions $c_1, \ldots, c_5$, which are defined below:

c_1 One of the leading processors has value [–,pref-a] or [–,a] and no leading processor has [–,pref-b].

c_2 One of the leading processors has [–,pref-b], or all the leading processors have value [–,b].

c_3 One of r_2 or r_3 is in the range [8,–] to [1,–] and the other is [2,–] or [3,a] or [3,pref-a].

c_4 One of r_2 or r_3 is in the range [8,–] to [1,–] and the other is [3,b] or [3,pref-b].

c_5 One of r_2 or r_3 is in the range [8,–] to [1,–].

Terminating Conditions: In above only nonterminating transitions (except in c_5) are specified. Now we give terminating conditions.

T_1 Each processor goes to [dec-a] ([dec-b]) whenever reads [dec-a] ([dec-b]).

T_2 If a processor is in [–,a] ([–,b]) and it sees that the other two processors are at least 2 steps behind, then it goes to [dec-a] ([dec-b]).

T_3 We assume that the registers also have a third field which is changed only when a processor gets out of each section ([8,-],[3,-]), or ([2,-],[6,-]), or ([5,-],[9,-]), it writes one of the following 3 values: (i) "a": Within this section, it always had [-,a], never had [-,b]; (ii) "b": Always had [-,b]; (iii) "c": Had both [-,a] and [-,b]. When all the processors are out of a section, if they all have "a" ("b") then go to [dec-a] ([dec-b]). This checking is trivial. Note that we ignored this third field in above in order to keep things readable.

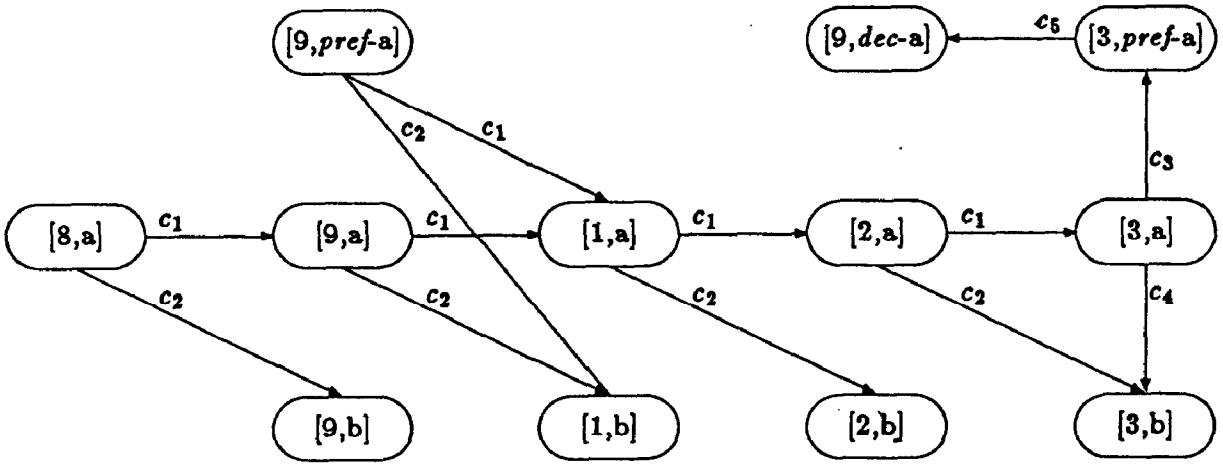

Figure 3: The three processor bounded Protocol (for P_1):

References

[1] M. Ben-Or, "Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols", *Proc. 2nd Annaul ACM Symposium on Principles of Distributed Computing (1983)* , pp. 27-30.

[2] G. Bracha and S. Toueg, "Asynchronous Consensus and Broadcast Protocols", *JACM* 32(4) (1985), pp. 824-840.

[3] E. Dijkstra, "Solution of a Problem in Concurrent Programming Control", *CACM* 8, p. 596 (1965).

[4] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process", *JACM* Vol. 32, No. 2, pp. 374–382 (1985).

[5] L. Lamport, "On Interprocess Communication", manuscript, DEC SRC, December 1985.

[6] M. Rabin, "The Choice Coordination Problem", *Acta Informatica* 17, pp. 121-134 (1982).

[7] D. Lehmann and M. Rabin, "On the Advantagesof Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem", *Proc. 8th Principles of Programming Languages* (1981) pp. 133-138.

Acknoledgements

The second author wishes to thank Prof. Nissim Francez and Dr. Lenny Pitt for disscusions that helped him start this work.