The Las-Vegas Processor Identity Problem (How and When to Be Unique)

EXTENDED ABSTRACT

Shay Kutten[*]
T.J. Watson Research Center
IBM

Rafail Ostrovsky[†]
UC Berkeley and
ICSI

Boaz Patt-Shamir[‡]
Lab for Computer Science
MIT

Abstract

One of the fundamental problems in distributed computing is how identical processes with identical local memory can choose unique IDs provided they can flip a coin. The variant considered in this paper is the asynchronous shared memory model (atomic registers), and the basic correctness requirement is that upon termination the processes must always have unique IDs.

We study this problem from several viewpoints. On the positive side, we present the first Las-Vegas protocol that solves the problem. The protocol terminates in (optimal) $O(\log n)$ expected time, using $O(n)$ shared memory space, where n is the number of participating processes. On the negative side, we show that there is no Las-Vegas protocol unless n is known precisely, and that no finite-state Las-Vegas protocol can work under schedules that may depend on the history of the shared variable. For the case of arbitrary adversary, we present a Las-Vegas protocol that uses $O(n)$ unbounded registers.

1 Introduction

Background and Problem Statement. In the course of designing a distributed algorithm, one has often to confront the problem of how to distinguish among the different participating processes. One attractive (and sometimes realistic) solution is simply to deny the existence of a problem altogether: assume that each process is fabricated with a unique identifier. This assumption seems especially reasonable where the processes are physical processors, and an ID which is guaranteed to be unique is "hardwired" in them (see, for example, the IEEE 48 Bit Standard [Tan81]). In some cases, however, one cannot get away with this approach. For instance, the processes might be virtual (i.e., software) processes that are spawned using the same code, and thus they are created identical. In this setting, the processes typically run on the same physical machine, and the communication among them is realized via shared memory space. The problem of giving distinct names to identical processes in such an environment is known by the name "Processor Identity Problem" [LP90]. In the sequel we denote it by PIP for short. We remark that PIP, due to its fundamental role, has applications in some of the most basic distributed tasks, including mutual exclusion, choice coordination, and resource allocation.

The formal specification of the problem is as follows. We have a system with n processes, and for each process there is some designated private output register. We assume that the output register is capable of storing $N \geq n$ different values. A protocol for PIP must satisfy the following conditions.

Symmetry: The processes execute identical protocols with identical initial state.

Uniqueness: Upon termination, all the values in the output registers are distinct.

In this paper, we shall study the problem mainly in the *read-write registers* model [Lam86], where in a single action, a processor can either read or write the contents of a single register and perform some local (probabilistic) computation. (This model is also known by the name of "atomic registers".) Clearly, in this model no deterministic protocol can solve PIP. For randomized protocols, we need a refined correctness requirement. Consider, for instance, the *Monte-Carlo* model, where the algorithm may fail with some low probability. If we are willing to tolerate erroneous termination, then there exists a trivial solution that does not require any communication: each process chooses independently a random ID; the error probability is controlled by the size of the ID space. Many applications, however, cannot tolerate any probability of error. In this paper we focus exclusively on the case where erroneous termination is outlawed. For this variant, a Monte-Carlo solution may guarantee that with

[*] E-mail: kutten@watson.ibm.com.

[†] E-mail: rafail@melody.berkeley.edu. Supported by an NSF postdoctoral fellowship and ICSI. Research partly done while visiting IBM T.J. Watson Research Center.

[‡] E-mail: boaz@theory.lcs.mit.edu. Supported in part by DARPA contracts N00014-92-J-4033 and N00014-92-J-1799, ONR contract N00014-91-J-1046, and NSF contract 8915206-CCR. Research partly done while visiting IBM T.J. Watson Research Center.

0-8186-3630-0/93 $3.00 © 1993 IEEE

high probability, the protocol terminates and provides unique IDs; in the case of failure, the protocol may never terminate.

The most restrictive probability model is the *Las-Vegas* model, in which not only the protocol is required to produce a correct answer always, but also its running time should have finite expected value. In this paper we shall study the necessary and sufficient conditions for the existence of Las-Vegas protocols for PIP.

Previous Work. Symmetry breaking is one of the well-studied problems in the theory of distributed systems. For example, see [Bur81, JS85, CIL87, ABD+87]. The Processor Identity Problem where errors are forbidden was first defined by Lipton and Park in [LP90]. Their formulation has two additional requirements. First, that the initial state of the shared memory is arbitrary (the "dirty memory" model); and second, that upon termination, the IDs constitute exactly the set $\{1,\ldots,n\}$. In [LP90], Lipton and Park also give a Monte-Carlo solution that for any given integer $L \geq 0$ uses $O(Ln^2)$ shared bits and terminates in $O(Ln^2)$ time with probability $1 - c^L$, for some constant $c < 1$. This protocol is Monte-Carlo, since failure results in an execution that never terminates. The protocol of [LP90] was subsequently improved by Teng [Ten90]. His protocol uses $O(n \log^2 n)$ shared bits, and gets, with probability $1 - 1/n^c$ (for any constant $c > 0$), running time of $O(n \log^2 n)$. As in [LP90], however, in the event of failure, the protocol never terminates. Recently, Eğecioğlu and Singh [ES92] have obtained independently a Las-Vegas protocol for PIP that works in $O(n^7)$ expected time using $O(n^4)$ shared bits.

Our Results and Organization of the Paper. In this work, we give tight positive and negative results describing the conditions under which the Las-Vegas Processor Identity Problem can be solved. First, we show how to solve the problem: in Section 3 we give the first Las-Vegas protocol for PIP in the read-write registers model. This protocol improves on all known protocols simultaneously in termination probability, running time, and shared space requirement. Specifically, the protocol terminates in $O(\log n)$ expected time, using $O(n)$ shared memory space.[1] As in the original formulation of PIP, the protocol has the nice features that it works in the dirty memory model, and that upon termination the given names constitute exactly the set $\{1,\ldots,n\}$. The protocol works under the assumptions that n, the number of participating processes, is known, and that the schedule is *oblivious* (i.e., the order in which processes take steps does not depend on the actual execution). We also consider the *dynamic* case, where processes may join and leave the system dynamically. Here, we relax the problem specification and require only that if no process joins the system for sufficiently long time, then eventually the IDs stabilize on unique values. We give a simple variant of our protocol for this case, which stabilizes in $O(\log n)$ expected time.

Next, we prove that the assumptions of Section 3 are actually necessary to obtain a Las-Vegas protocol for PIP. Specifically, in Section 4 we prove the following results. First, we prove that there is no Las-Vegas protocol that solves PIP if only a *bound* on n is known in advance, even if the schedule is oblivious. We also show that any protocol for PIP must run in $\Omega(\log n)$ expected time if errors are disallowed. Most interestingly, we show that even if n is known, there is no finite-state Las-Vegas protocol for PIP when the schedule is allowed to be *adaptive*, i.e., when an adversary can decide which process moves next based on the history of the shared variables. This impossibility is complemented with a protocol for PIP that works for any fair scheduler — using unbounded space. The interpretation of these results is that in the Las-Vegas model, one can have either a bounded space protocol, or a protocol which is resilient to arbitrary adversaries, but not both.

Finally, in Section 5, we consider the *read-modify-write* model, in which a processor can, in a single indivisible step, access a register and update its value in a way that may depend on the current contents of that register. We show some similarities and some deep differences between this model and the model of read-write (i.e., atomic) registers. Specifically, we show that in this model PIP can be solved *deterministically* using only 3 shared bits under any fair adversary, if the system is initialized properly. On the other hand, there is no finite-memory randomized protocol that solves PIP if the initial state is arbitrary and the adversary is adaptive.

We start with a brief description of the models we consider in Section 2.

2 The Model

Our basic model is the asynchronous shared-memory distributed system. Such a system is characterized by a set of n *processes* denoted by $p_1,\ldots,p_n$, and m shared *registers*, or *variables*, denoted by $r_1,\ldots,r_m$. The processes are modeled as probabilistic state machines, and the shared registers may hold values from some specified domain. All processes can read and write all the shared registers. A *state* of the system is completely determined by the states of the individual processes and of the shared registers. The state may be altered by processor *actions*. We assume that in every state, at most one action is *enabled* in each process.

Action Models. In this paper we shall be concerned mainly with the *read-write* model, where each action consists of an access to the shared memory (which is either a read or a write of a shared variable), and some (possibly randomized) local computation. We shall also discuss the *read-modify-write* model, where a process can, in a single indivisible step,

[1] We remark that the protocol presented in this paper uses registers of size $O(\log n)$ bits, whereas the protocol of [ES92] uses single bit registers.

obtain the value of a shared variable, rewrite that variable as a function of the process local state and the current value, and perform some local (probabilistic) computation.

Asynchronous Executions and Adversaries. Following the IO Automata model of Lynch and Tuttle, we model an *execution* of a system as an alternating sequence of states and actions, where in each step, one process makes an action to yield the next state (cf. [LT89]). To model asynchrony, we assume that the choice of which process takes the next step is under the control of an adversary. There are two types of adversaries we consider in this work. The *oblivious adversary* is simply an infinite sequence of processes names. The *adaptive adversary* is defined by a function that takes a finite sequence of shared memory states, and produces a process name. Intuitively, the oblivious adversary commits itself to the order in which processes are scheduled to take steps oblivious to the actual execution, while the adaptive adversary may observe the shared memory and choose which process to schedule next based on this knowledge. The only general restriction we impose on the adversaries is that they must be *fair*, i.e., (in our context) each process must be scheduled to take steps infinitely many times.

Time Complexity. To measure time in the asynchronous model, we assume that all processes take at least one step (either a read or a write) in each time unit. The executions are completely asynchronous, i.e., the processes have no access to clocks, and the "real time" notion is assumed here only for the purpose of analysis.

3 A Solution for the Processor Identity Problem

In this section we give a protocol that solves the Processor Identity Problem in the asynchronous shared memory model. Our protocol uses $O(n)$ shared bits, and terminates in $O(\log n)$ expected time, where n is the number of participating processes. We solve the problem in its original formulation [LP90], i.e., assuming the dirty memory model, and producing as the final names exactly the set $\{1, \ldots, n\}$. We remark that our protocol relies on the assumptions that n is known in advance, and that the schedule is oblivious. (We show in Section 4 that both assumptions are necessary to ensure termination with probability 1.)

3.1 A Las-Vegas Protocol for PIP

We begin with an intuitive description of the protocol. In the execution of the protocol, each process proceeds in a loop as follows. At every given point in the execution of the loop, each process *claims* some "tentative" ID, which it repeatedly checks to verify that no other process claims. If a process detects a competitor for its claimed ID (we call this case a *collision*), the process chooses a new ID at random. The loop terminates once the process can safely deduce that all processes claim distinct IDs.

The protocol is based on a few implementation ideas, designed to solve the problems of collision detection and termination detection. To ease exposition, we first assume that the memory is initialized with a special value Λ. We then explain the extensions that enable the algorithm to work with dirty memory, and get the final IDs to be $\{1, \ldots, n\}$. The full protocol is presented in Figure 1.

Collision detection (lines 1-8). To detect an ID claimed by more than a single process, the protocol executes repeated checking as follows. We have in the shared memory a vector of N registers, where $N = c \cdot n$ for some constant $c > 1$. Each register corresponds to a particular tentative ID, namely the index of that register. During a checking phase, a process either reads the register corresponding to its tentative ID, or writes it. The choice between the alternatives is random. If the process writes, it writes a random *signature*: a signature is simply a random bit, which is drawn independently anew whenever the process signs. The value of the last signature is recorded in the local memory. If the process chooses to read, it examines the contents of the register to see if some other process has changed it since its last own write. If a different signature is detected, then the process concludes that its claimed ID gives rise to a collision. In this case the process chooses a new tentative ID uniformly at random from $\{1, \ldots, N\}$, and then proceeds with this new ID to the next iteration. If no change was detected, then the process proceeds to the next iteration.

Termination detection (lines 9-18). It is important to observe that once an ID is claimed, it will remain claimed: at any point, the last process to write a random signature at the corresponding register claims that ID. In other words, the set of claimed IDs is monotonically increasing. This property of the algorithm facilitates the termination detection mechanism, that works parallel to the collision detection mechanism. The idea is to count the number of claimed IDs; the monotonicity property ensures that a process can only underestimate the number of claimed IDs, and thus, when the estimate is n, the collision detection can stop. We shall also guarantee that after the set of claimed IDs has stabilized, eventually all the claimed IDs will be detected as such. The basic idea in counting the number of claimed IDs is to fan-in the local sums is a "tree addition" fashion. Specifically, we use the following implementation strategy. The vector of registers used for the collision detection is treated as "level 0" (i.e., the leaves) of a tree, denoted $\mathcal{D}[i, 0]$, where $1 \leq i \leq N$ is the index that corresponds to a tentative ID. The other levels are defined in the natural way as follows. For each "leaf" register $\mathcal{D}[i, 0]$ we define its *ancestor register* at level j, for all $0 \leq j \leq \log N$, by

$$ancestor(i, j) = \mathcal{D}\left[1 + \left\lfloor \frac{i-1}{2^j} \right\rfloor , j\right] .$$

Shared Variables

$\mathcal{D}$: *a complete binary tree of height* $\lceil \log N \rceil$*; registers are indexed by location and level*

Local Variables

$tent\_ID$: *tentative ID, initially* $random(\{1, \ldots, N\})$

$signature$: *takes values from* $\{0, 1, \Lambda\}$*, initially* Λ *(retains state of claimed register)*

ID : *output value*

$level$: *takes values from* $0 .. \log N$*, initially 0* *(current level of responsibility for summation)*

seg_left,
seg_right,
seg_tot : *take values from* $0..N$ *(for counting number of claimed IDs in segment)*

M_image : *local view of the status of the shared memory locations, initially all* `dirty`

Shorthand

$random(S)$: *returns a random element of* S *under the uniform distribution.*

$$\langle v \rangle = \begin{cases} 1, & \textit{if level} = 0 \textit{ and } v \neq \Lambda \\ v, & \textit{if level} \neq 0 \textit{ and } v \neq \Lambda \\ 0, & \textit{if } v = \Lambda \end{cases}$$

$\text{WRITE}(r, v) \equiv [M\_image(r) \leftarrow \texttt{clean};\ contents(r) \leftarrow v]$ *(write v in register r)*

$\text{READ}(r) \equiv [\textbf{if } M\_image(r) = \texttt{dirty} \textbf{ then } \text{WRITE}(r, \Lambda)];\ [\textbf{return } contents(r)]$

Code

```
1   repeat
2     either, with probability $1/2$, do                                  (write and record random signature)
3       $\mathcal{D}[tent\_ID, 0] \leftarrow signature \leftarrow random(\{0,1\})$
4     or, with probability $1/2$, do                                       (read and check for collision)
5       if $\mathcal{D}[tent\_ID, 0] \notin \{signature, \Lambda\}$ then
6         $tent\_ID \leftarrow random(\{1, \ldots, N\})$
7         $signature \leftarrow \Lambda$
8         $level \leftarrow 0$
9     if $level > 0$ then                                          (write sum of entries from lower level segment)
10      $ancestor(tent\_ID, level) \leftarrow seg\_tot$
11    if $level < \log N$ then                                      (read the segment in the current level)
12      $seg\_left \leftarrow$ left entry of segment of $ancestor(tent\_ID, level)$
13      $seg\_right \leftarrow$ right entry of segment of $ancestor(tent\_ID, level)$
14    if $ancestor(tent\_ID, level)$ is the first non-$\Lambda$ entry in the segment and $level < \log N$ then   (verify responsibility)
15      $seg\_tot \leftarrow \langle seg\_right \rangle + \langle seg\_left \rangle$    (sum up number of claimed IDs in segment)
16      $level \leftarrow (level + 1) \bmod \lceil \log N + 1 \rceil$
17    else $level \leftarrow 0$                                          (start summing from the leaf again)
18  until $\mathcal{D}[1, \log N] = n$                                          (termination predicate)
19  $ID \leftarrow |\{i : \mathcal{D}[i, 0] \neq \Lambda \text{ and } i \leq tent\_ID\}|$   (compute rank of tent_ID by parallel prefix)
```

Figure 1: Terminating algorithm for PIP in the dirty memory model, with IDs in the range $\{1, \ldots, n\}$. Accesses to the shared memory in lines 1-18 are interpreted by the READ and WRITE definitions.

The ancestor relation defines, in the obvious way, a directed tree whose root is $\mathcal{D}[1, \log N]$. The semantics of this tree is that each entry $\mathcal{D}[i, \mathit{level}]$ is intended to contain the number of claimed IDs in the subtree rooted at that entry (i.e., all the IDs in the range $(i-1)2^{level}+1 \ldots i \cdot 2^{level}$). For level 0, any random signature counts as 1; only Λ counts as 0 (see the $\langle\cdot\rangle$ notation). This is done by the following rule. The registers at each level $j < \log N$ are paired by the *sibling relation*, that is, pairs of entries with common ancestor in level $j+1$. Call these pairs *segments*; a segment is said to be *active* at a given state if it contains an *ancestor* of some claimed ID. The tree summation proceeds by assigning the "responsibility" for each active segment to the "left-most" process in its subtree. The responsible process fills the parent entry with the total number of claimed IDs rooted in that parent. More specifically, each process reads the entry corresponding to its *ancestor* and its ancestor's sibling (lines 12-13); the process deems itself responsible if either its ancestor's sibling is Λ, or if the ancestor's index is smaller than the sibling's (line 14). If the process determines that it is responsible for that segment, then in the next iteration it will write the sum of the entries in that segment in the parent, and will proceed to find whether it is responsible for the higher segment. Notice that due to the asynchrony, many processes may claim responsibility of the same segment, thus overwriting the same entry. To overcome this problem, the processes refresh their entries in a cyclic manner: once a process reaches the level for which it is no longer responsible, it start working again all the way from level 0 up (line 17), and thus the values under its responsibility are re-written at least once every $O(\log n)$ time units. We shall show that this procedure guarantees that once all the collisions disappear, eventually the root of the tree (in our notation, $\mathcal{D}[1, \log N]$) will contain n. Moreover, at any point before all the collisions has been resolved, no process reads $\mathcal{D}[1, \log N] = n$.

Renaming the processes (line 19). After the algorithm has created n distinct IDs, it is straightforward to rename the processes so that their names constitute the set $\{1, \ldots, n\}$ in additional $O(\log n)$ time. To do that, notice that if the root of the summation tree becomes n, then clearly all the accessed values in the tree have stabilized to their intended meaning. Our method is simply to assign to each process the *rank* of its tentative ID, i.e., its index in the sorted list of the claimed IDs. This can be done easily using the existing tree: apply some kind of standard parallel "prefix sums" algorithm (see, e.g., [Lei91]). We remark that the processes need only work on active segments, and hence the running time is always $O(\log N)$, even if $n \ll N$ (cf. dynamic protocols).

Read when clean (shorthand notations). The algorithm as described above works under the assumption that all the memory entries are initialized with a special value Λ. One naive approach to make the algorithm work in the dirty memory model is to first let all processes initialize the whole memory with Λ, and disregard Λ values while computing (e.g., detecting Λ is not considered a collision in line 5). Informally, the correctness is maintained since each meaningful value is re-written periodically in the main loop. This method, although correct, is clearly much too wasteful: everyone erase all locations, resulting in $\Omega(n)$ running time.

A simple fix, however, reduces the time complexity dramatically. We introduce a rule called "read when clean", defined as follows. Each process, locally, keeps track of the status of each register of the shared memory. Initially, all registers are marked *dirty* at all the processes; whenever a process writes, it updates its local record by marking the written register *clean* . The rule observed by the algorithm is that a process reads only clean registers; this rule is maintained by erasing the contents of a dirty register when it is to be read. More specifically, whenever a register needs to be read, its status is checked; if it is *dirty*, then the special value Λ is written, and only then the register is read. This rule is used only in the main loop; after the values have stabilized (i.e., in line 19), no erasures are made.

The basic property of the read-when-clean rule is that a register is erased only if it is necessary. For example, if a subtree does not contain any claimed ID, then none of its nodes (except the root, possibly) is accessed.

3.2 Analysis

We now state the correctness of the protocol given in Figure 1. Due to lack of space, we only sketch the main arguments of the proof.

Theorem 1 *The algorithm in Figure 1 produces, upon termination, unique IDs in the range* $\{1, \ldots, n\}$*. Moreover, it terminates with probability* 1*, and its expected termination time is* $O(\log n)$.

We call a tentative ID value *unique* if at most one process claims it. For the following lemma, recall that $N = c \cdot n$.

Lemma 1 *Whenever a process chooses a new value for tent_ID, this value is unique with probability at least* $1 - \frac{1}{c}$.

The next lemma states the essential property that guarantees the correctness of the termination detection mechanism.

Lemma 2 *If an ID is claimed at some state of the execution, then it remains claimed from that point on.*

Proof Sketch: The lemma is immediate for IDs that are unique for the remainder of the execution. Now suppose that two or more processes claim some ID. Observe that by the code, whenever a collision is detected, the value in the register is not changed, and that a process may change its *tent_ID* only after reading. Therefore, at any time, the last process to write the register claims that ID. ▮

The following lemma shows that when a process exits the main loop, there are n distinct claimed IDs.

Lemma 3 *If a process reads some value v at some $\mathcal{D}[i, j]$ for $j \geq 1$, then the number of claimed IDs in the range $(i-1) \cdot 2^j + 1, \ldots, i \cdot 2^j$ at that time is at least v.*

Proof Sketch: By induction on the levels. Consider $\mathcal{D}[i, 1]$, for $1 \leq i \leq N/2$. If a process reads some value v at that location, then the "read when clean" rule ensures that some process wrote that value v. For level 1, this means that some process has detected that v of the IDs $\{2i-1, 2i\}$ are claimed. By lemma 2 and the "read when clean" rule, there are indeed at least v claimed IDs at that segment at any point after v was written at $\mathcal{D}[i, 1]$. A similar argument can be applied to higher levels as well. ▮

We now state the main lemma needed for the correctness of the collision detection mechanism.

Lemma 4 *Suppose $k > 1$ processes claim the same ID at some state at time $t \geq 1$. Then there exist constants $p, \mu > 0$ such that in the following 18 time units, at least μk of these processes choose a new $tent\_ID$ with probability at least $1 - p^k$.*

Proof Sketch: If one of the processes has terminated, then by Lemma 3 we are done. So suppose now that none of the processes has terminated. Notice that a process completes a loop in at most 9 time units (counting read as two steps, due to the read-when-clean mechanism). Consider the time interval $[t+9, t+18]$. Let k' be the number of processes that claimed the ID and accessed its register in that interval. If $k' \leq (1-\mu)k$, we are done.

We now focus on the case where $k' > (1-\mu)k$ processes accessed the variable in $[t+9, t+18]$. Denote the sequence of accesses to the claimed variable in this time interval by $a_1, a_2, \ldots$, and denote the first access of process p_j by a_{i_j}, for $1 \leq j \leq k'$. Consider the pairs of events $a_{i_j}, a_{i_j - 1}$, for all j such that $a_{i_j - 1}$ is defined. In other words, we consider the first access of p_j and the access *immediately preceding it*. Note that $a_{i_j - 1}$ is an action of some process other than p_j, by definition of a_{i_j}. Now, consider the following events:

$$a_{i_2 - 1}, a_{i_2}, a_{i_4 - 1}, a_{i_4}, \ldots, a_{i_{2\lfloor k'/2 \rfloor} - 1}, a_{i_{2\lfloor k'/2 \rfloor}} .$$

We first claim that all these events are distinct. This is easy to see since between $a_{i_{2j}}$ and $a_{i_{2(j+1)}}$ there must occur $a_{i_{2j+1}}$. We now argue that p_{2j} chooses a new $tent\_ID$ with some constant probability $p' > 0$, for $1 \leq j \leq \lfloor k'/2 \rfloor$. To see this first notice that p_{2j} does not perform an erasure action since it has already accessed the the register in the previous iteration of its main loop. Also, with some positive probability $a_{i_{2j} - 1}$ is done by another process, that wrote a random signature different than p_{2j}'s, and p_{2j} reads it. The crucial observation here is that the obliviousness of the protocol guarantees that all these occur *independently*, and therefore the probability of the intersection is a positive constant. Finally, we argue that the $\lfloor k'/2 \rfloor$ events of processes p_{2j} detecting a collision are also independent, which follows from the fact that all the events involved are distinct. The result follows by applying Chernoff bound (see, e.g., [AS91]). ▮

Using Lemma 4, we obtain the following result.

Lemma 5 *After $O(\log n)$ expected time units, the number of claimed IDs is n.*

Theorem 1 is a direct corollary of Lemmas 5, 3 and the following easy lemma.

Lemma 6 *If the protocols reaches a state with n claimed IDs, then in $O(\log n)$ time units all processes exit the main loop with correct values in accessed tree registers.*

To conclude the analysis of Algorithm 1, we remark that the size of the shared space (in bits) is

$$2N + \sum_{level=1}^{\log N} \frac{(level+1)N}{2^{level}} < 5N = O(n) .$$

The first term is for the level 0 registers (each of size 2 bits). Each register at level $level$, for $level > 0$, needs to hold values in the range $0..2^{level}$, which implies size of $level + 1$ bits. The number of registers at level $level$ is $N/2^{level}$.

3.3 Dynamic Protocols

Consider the *dynamic setting*, where processes may join and leave the system dynamically. In this case, the protocol only has a *bound* on the number of processes that may be active simultaneously. As proved in Section 4, such a protocol cannot terminate (Theorem 3). We therefore relax the correctness requirement as follows. We require that if processes stop joining the system, then the IDs in the output registers will eventually *stabilize* on distinct values.

The algorithm of Figure 1 can serve as the basis for efficient dynamic protocols as follows. First, we can simply repeat the main loop forever, and interleave in it repeated rank calculation (i.e., performing one step of the parallel prefix computation in each iteration of the main loop). This yields a dynamic protocol that stabilizes in $O(\log N)$ time on the names $\{1, \ldots, n\}$, for the actual number n of active processes so long as $N \geq c \cdot n$ for some $c > 1$.

Another simplification can be done if we do not care that the IDs will be exactly the set $\{1, \ldots, n\}$. Specifically, we can get rid of the counting mechanism altogether. Moreover, there is no need for the "read when clean" rule any more. The algorithm then reduces only to an infinite collision detection procedure. The simplified algorithm is given in Figure 2. We remark that the algorithm is, in fact, self-stabilizing: the IDs will eventually stabilize on unique values regardless of the initial state of the system. The correctness of the algorithm is stated in the theorem below.

Theorem 2 *The algorithm in Figure 2 stabilizes in $O(\log N)$ expected time units with all IDs unique, assuming that $N \geq c \cdot n$ for some $c > 1$.*

The detailed analysis of the algorithm uses ideas similar to the analysis of the algorithm of Figure 1 and is omitted.

```
Shared Variables
  $\mathcal{D}$            : a vector of N bits

Local Variables
  $signature$    : a bit
  $ID$           : output value

Code
repeat forever
  either, with probability $1/2$, do
    $\mathcal{D}[ID] \leftarrow signature \leftarrow random(\{0,1\})$
  or, with probability $1/2$, do
    if $signature \neq \mathcal{D}[ID]$ then
      $ID \leftarrow random(\{1,\ldots,N\})$
      $signature \leftarrow \mathcal{D}[ID]$
```

Figure 2: Simple dynamic algorithm for PIP.

4 Necessary Conditions for Solving PIP

In this section we show that in the read-write model, no terminating algorithms exist for PIP if either n is not known in advance, or if the schedule is adaptive. We shall also argue that there is no protocol for PIP that terminates in $o(\log n)$ time. All these impossibility results are based on the observation that at least one of the processes needs to "communicate" (not necessarily directly) with all the other processes before termination. We remark that this observation translates fairly easily into rigorous proofs of the time lower bound and the necessity of knowledge of n; the proof of impossibility for adaptive adversaries is more involved.

We analyze the protocols for PIP in terms of Markov chains [Fel68]. Consider a single process taking steps according to a given PIP protocol. (Even though we might have $n > 1$, we consider only one process taking steps without interference of other processes.) That process can be viewed as a Markov chain, whose state is characterized by its local state and the state of the shared memory. We shall represent that Markov chain as a directed graph, whose nodes are the states of the Markov chain, and a directed edge connects two nodes iff the probability of transition from one to the other is positive. Given a protocol $\mathcal{P}$ for PIP, this Markov chain is completely determined. Note that the symmetry condition implies that all processes have identical Markov graphs. In the sequel, we denote the graph corresponding to a protocol $\mathcal{P}$ by $G_{\mathcal{P}}$, and we shall use the terms "states" and "nodes" interchangeably. This graph has possibly many nodes with no incoming edges (called hereafter *source nodes*), that correspond to the possible initial states of the system. All these states have the same local portion, but they may differ in the state of the shared memory, due to the dirty memory assumption.

We start with a general lemma for PIP in the read-write model.

Lemma 7 *Let s be any reachable node in $G_{\mathcal{P}}$, and let s^* be the global state of the n-process system with the same shared state portion as in s, and such that the local states of all processes are identical to the local state portion in s. Then there exists an oblivious schedule under which the system reaches s^* with positive probability.*

Proof: We shall show that the "round robin" schedule works. Let s_0 be a source node in $G_{\mathcal{P}}$ from which s is reachable. We prove the lemma by induction on the distance d of s from s_0 in $G_{\mathcal{P}}$. The base case, $d = 0$, follows (with probability 1) from the symmetry requirement for PIP: s^* is the state where the shared memory is in the same state as in s, and all the processes are in the same state as in s. For the inductive step, suppose that s is at distance $d + 1$ from s_0. Let s' be the node in $G_{\mathcal{P}}$ which is reachable in d steps from the s_0, and such that s is reachable in one step from s'. By the inductive hypothesis, the global state corresponding to the symmetric combination of the s' nodes is reachable with some probability $\alpha > 0$. By the definition of $G_{\mathcal{P}}$, s is reachable in a single step of process p_i, with some probability $\beta > 0$, for all $1 \leq i \leq n$. Now consider scheduling exactly one step of each process. Since the processes take their steps when they are in identical local states, it must be the case that they either all read or all write in their respective additional step. Hence, their steps cannot influence one another, which means that the probability distributions of their next state are *independent*. Therefore, with probability $\alpha\beta^n > 0$, the global state s^*, is reached, and the inductive step is complete. ▌

Notice that Lemma 7 holds even for infinite state protocols (so long as no zero probability transitions are ever taken).

Using Lemma 7, we can now prove the first necessary condition for Las-Vegas PIP protocols. We shall prove a slightly stronger result, namely that even termination with probability 1 is impossible in these conditions.

Theorem 3 *There is no protocol for PIP that terminates with probability* 1 *and works with unknown number of processes, even for oblivious schedules.*

Proof: By contradiction. Suppose, for simplicity, that $\mathcal{P}$ works for $n = 1$ and $n = 2$. (The argument extends directly to arbitrary different values of n.) We argue that in this case, $\mathcal{P}$ cannot terminate when run with a single process. For suppose not: let ρ be any terminating execution in which only one process takes steps. By Lemma 7, there exists an execution ρ' of positive probability with two processes such that both processes reach the same state as in the end of ρ. But since the last state in ρ is a terminating state, we conclude that both processes terminate in ρ', and since they are in identical local state, they must have the same output value, a contradiction to the uniqueness requirement. ∎

Again, we remark that Theorem 3 holds also for infinite memory protocols.

We now turn to consider the case of adaptive adversary. Intuitively, the adaptive adversary picks the processes to take steps based on the history of the system, or more precisely on the history of the shared portion of the system. (The adversary has no access to in local state of the processes.) The following theorem implies that there is no finite-state Las-Vegas protocol for adaptive adversaries. Again, we prove that no protocol for PIP can terminates with probability 1 under these conditions.

Theorem 4 *There is no finite protocol for the Processor Identity Problem that terminates with probability* 1 *if the schedule is adaptive, even if n is known.*

Proof: By contradiction. Suppose that a given protocol $\mathcal{P}$ terminates with probability 1. Then for all $\epsilon > 0$ there exists T_ϵ such that the probability of $\mathcal{P}$ terminating in T_ϵ or less time units is at least $1 - \epsilon$. We shall derive a contradiction by showing that there exists $\epsilon_0 > 0$ (that depends only on $\mathcal{P}$ and n), such that for any given time T, there exists an adaptive schedule in which $\mathcal{P}$ cannot terminate in T time units with probability greater than $1 - \epsilon_0$.

The our strategy, as in the proof of Theorem 3, is to keep the processes "hidden" from each other. For simplicity of presentation, let us consider the case of $n = 2$. The proof can be extended to an arbitrary number of processes in a straightforward way.

Our first step is to decompose the corresponding Markov chain into irreducible chains. In graph theoretic language, consider the Markov graph $G_\mathcal{P}$: it is a directed graph; we decompose it into strongly-connected components. That is, we partition the nodes into equivalence classes ("strong components"), such that two nodes s and s' are in the same class if and only if there is a directed path in $G_\mathcal{P}$ from s to s' and from s' to s. Given this decomposition, we define a *terminal component* to be a strong component such that no other component is reachable from it. (We remark that terminal components are the irreducible components of the given Markov chain.) Notice that the existence of terminal components in the Markov graphs is guaranteed by the fact that the number of nodes in $G_\mathcal{P}$ (i.e., the number of states of the protocol $\mathcal{P}$) is finite. We now use a simple fact from the theory of Markov chains.

Lemma 8 *Let s be a state in a terminal component of $G_\mathcal{P}$, and let s^* be any global state that extends s with respect to some process p_i. Then for all $\gamma < 1$ there exists a positive integer M_γ, such that in an execution that starts at s^* and consists of M_γ steps of p_i alone, s^* occurs again with probability at least γ.*

Proof: The state of a process p_i in an execution that starts from a state in a terminal component, and in which only p_i takes steps, is an irreducible Markov chain. The lemma follows from the fact that for a finite irreducible Markov chain, the expected recurrence time of any state is finite. ∎

Lemma 8 gives rise to the following immediate corollary.

Corollary 9 *Let s be a state in a terminal component of $G_\mathcal{P}$, let s^* be any global state that extends s with respect to some process p_i, and let $\bar{v}$ be the state of the shared portion of s^*. Then for all $\gamma < 1$ there exists a positive integer M_γ, such that in an execution that starts at s^* and consists of M_γ steps of p_i alone, $\bar{v}$ occurs again with probability at least γ.*

We now continue with the proof of Theorem 4, by describing a complete strategy of an adaptive adversary, given a protocol $\mathcal{P}$ and a time bound T. First, an arbitrary reachable state s in a terminal component of $G_\mathcal{P}$ is chosen. (This can be done since the protocol completely characterizes $G_\mathcal{P}$.) The adversary then uses the round-robin schedule outlined in the proof of Lemma 7, which guarantees, with some probability $\alpha > 0$, that the corresponding symmetric global state s^*, in which all the processes are in the same local state as in s, and the shared memory is in the same state as the shared portion of s. We shall show that for any given time T, there is an adaptive schedule such that the protocol fails to terminate in T time units with probability greater than $\alpha/5$.

We do this as follows. Let $\gamma = 1 - 1/2T$. Denote the state of the shared portion in s^* by $\bar{v}$. The adversary now lets p_1 take steps (at least one) until the values of the shared variables are again as specified by $\bar{v}$, or until M_γ steps (as obtained from Corollary 9) have been taken. This can be done since by our assumption that the adversary is adaptive, the adversary can "see" when $\bar{v}$ recurs. Moreover, Corollary 9 guarantees that this process will succeed with probability at least γ. When $\bar{v}$ recurs, the adversary lets p_2 take steps (again, at least one and no more than M_γ), until the configuration of the shared memory is $\bar{v}$ again. This can be done by the same reasoning as for p_1. Notice that now, p_1 and p_2 are *not* necessarily in the same state; however, p_2 is completely "hidden" from p_1, since the shared memory is in exactly the same state in which p_1 stopped taking steps. Therefore, the adversary can resume p_1 now, and p_1 must act as if it is the only process in the system.

Observe that this procedure of letting one process take steps until $\bar{v}$ is reached and then switch to the other process, can be repeated $2T$ times (thus resulting in a schedule with running time T), with probability of success at least

$$\gamma^{2T} = \left(1 - \frac{1}{2T}\right)^{2T} > \frac{1}{5} \qquad \text{for } T \geq 1 \, .$$

We can now complete the proof of Theorem 4. We argue that for any given $T > 0$, using the schedule specified above we get, with probability at least $\epsilon_0 = \alpha/5$, an execution in which neither p_1 nor p_2 can terminate in T time units. This is since with probability at least α, s^* is reached, and with probability greater than $1/5$, once s^* is reached, p_1 and p_2 remain "hidden" from each other for T time units. Now we claim that termination of any of the processes in this execution implies a contradiction: since p_1 (say) did not observe any action of p_2, it follows that from the point of view of p_1, there exists an indistinguishable execution ρ in which p_2 is advancing in "lockstep" with p_1, maintaining symmetrical local state. If p_1 terminates in the given execution, then in ρ p_1 and p_2 terminate also, violating the uniqueness requirement. Since the uniqueness property must be met *always*, we have reached a contradiction. ▮

To show the necessity of the bounded space condition in Theorem 4, we have the following theorem.

Theorem 5 *There exists an unbounded-space algorithm for PIP that terminates with probability* 1 *under any fair adversary.*

```
Shared Variables
  $\mathcal{D}$          : a vector of $N$ integers, initially all 0

Local Variables
  ID         : output value, initially $random(\{1, \ldots, N\})$
  signature : an integer, initially 0

Code
repeat
    either, with probability $1/2$ do
        if $signature \neq \mathcal{D}[ID]$ then
            $ID \leftarrow random(\{1, \ldots, N\})$
            $signature \leftarrow 0$
    or, with probability $1/2$, do
        $\mathcal{D}[ID] \leftarrow signature \leftarrow$
                      $\leftarrow (2 \cdot signature + random(\{0, 1\}))$
until $| \{i \, : \, \mathcal{D}[i] \neq 0\} | = n$
$ID \leftarrow | \{i \, : \, \mathcal{D}[i] \neq 0$ and $i \leq tent\_ID\} |$
```

Figure 3: Unbounded space algorithm for PIP under any adversary.

The proof consists of an unbounded protocol; the protocol is a simple variant of our basic protocol, where the contents of each register is simply a complete history of all the random signatures. We present a simplified version of it in Figure 3.

Our last result for this section is the simple observation that any protocol for PIP requires $\Omega(\log n)$ time units.

Theorem 6 *There is no protocol for PIP (including Monte-Carlo protocols) that terminates in* $o(\log n)$ *expected time.*

Proof Sketch: We shall show that for any protocol there exists a schedule such that the expected running time under this schedule is at least $\log n$. First, for any given execution of a protocol, we define the set of ***influencing processes*** for a process p_i at state s_t, denoted $S_i(t)$, for all i and t. The set is defined inductively as follows. At the initial state, we define $S_i(0) = \{p_i\}$ for all i. Suppose now that $S_i(t')$ is defined for all the processes and all steps $t' < t$, and consider the action a_t leading to state s_t. If a_t is a write action, then $S_i(t) = S_i(t-1)$ for all i. If a_t is a read action of a process p_k, say, let p_j the last process that wrote that register (if such p_j exists), and suppose this write occurred at action a_w. In this case we define $S_k(t) = S_k(t-1) \cup S_j(w)$, and $S_i(t) = S_i(t-1)$ for all $i \neq k$. If no such p_j exists, define $S_i(t) = S_i(t-1)$ for all i. Intuitively, the influencing set of a process at a state is the set of all processes that have communicated with that process directly or indirectly.

We claim that in any given execution ρ of a protocol for PIP, a process may terminate only when its influencing set is $\{1, \ldots, n\}$. For suppose not, i.e., p_i terminates at step t with $p_j \notin S_i(t)$. Then there exists another positive-probability execution ρ', indistinguishable from ρ for p_i, in which p_j has the same random choices as p_i has, and in which p_i and p_j advance in lockstep. Clearly, p_i and p_j maintain identical local state in ρ', which in turn is identical to the state of p_i in ρ. Hence termination of p_i in ρ implies termination of p_i and p_j in ρ' with the same output value, a contradiction to the uniqueness requirement.

Now consider the executions resulting from the round robin schedule, where each step takes exactly one time unit. An easy induction on time shows that in these executions, $|S_i(t+1)| \leq 2|S_i(t)|$ for all processes p_i and steps t. Since we must have $|S_i(t)| = n$ at the terminating state for all processes, we conclude that the expected worst-case running time of every protocol for PIP is $\Omega(\log n)$. ▮

5 PIP and the Read-Modify-Write Model

In this section we give positive and negative results for PIP in the read-modify-write model. First, we show that in this model, PIP can be solved deterministically using constant size memory, under any fair schedule. This result should be contrasted with the impossibility results of Theorems 3 and 4 for the read-write model. Our second result for this section shows that if the initial state of the protocol is arbitrary (i.e., self-stabilization model), then there is no protocol (including randomized protocols), that solves PIP with probability 1. This result uses the technique of Theorem 4.

Let us start with an informal description of a protocol for PIP that uses $\log N$ bits. We will then derive our constant-space protocol. The protocol using $\log N$ bits is trivial: the shared variable is used as a counter, or a"ticket dispenser", in the following sense. When a process enters the system, it accesses the variable, takes its current value to be its ID, and in the same step, increments the value of the variable by 1. It is straightforward to verify that this indeed produces unique IDs at the processes under any fair schedule.

In our constant space protocol, we still employ this "serial counter" approach. However, to reduce space, we shall use the shared variable as a "pipeline" to transmit information from one process to another, while the relevant information is maintained in the *local memory* of the processes. Specifically, there will be some process "in charge" at any given time such that this process knows the current value of the counter. Whenever a new process enters the system, it writes a request message in the shared variable. The process in charge responds by transmitting the current contents of the counter, bit by bit, with an acknowledgment for each bit. By the end of this procedure, the new process has the value of the counter, it increments it by 1, takes it to be its ID, and becomes the process in charge. This serial style dialog will not be interrupted by other processes, by the read-modify-write assumption. Also, we assume that the shared variable is initialized with a special value, that tells whoever accesses the variable first, that it is in charge, and the counter value is 0. The formal specification of the algorithm is omitted from this abstract. We summarize in the following theorem.

Theorem 7 *In the read-modify-write model, there exists a deterministic protocol for PIP that requires a shared variable of 3 bits and works under any fair schedule.*

We now state the impossibility result for Las-Vegas PIP protocols in the read-modify-write model.

Theorem 8 *There is no finite state, self-stabilizing protocol that solves PIP in the read-modify-write registers with probability* 1 *if the schedule is adaptive and n is unknown.*

The proof is nearly identical to the proof of Theorem 4, and we therefore omit it. We remark only that the self-stabilization assumption serves as a substitute for Lemma 7, which does not hold in the read-modify-write model.

Acknowledgment

We thank Oded Goldreich for many insightful comments on an earlier draft of the paper. The third author would like to thank also Nancy Lynch and Yishay Mansour for many helpful discussions.

References

[ABD$^+$87] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, Daphne Koller, David Peleg, and Rüdiger Reischuk. Achievable cases in an asynchronous enviornment. In *28th Annual Symposium on Foundations of Computer Science, White Plains, New York*, pages 337–346, 1987.

[AS91] Noga Alon and Joel H. Spencer. *The Probabilistic Method.* Wiley Interscience, 1991.

[Bur81] James E. Burns. Symmetry in systems of asynchronous processes. In *21st Annual Symposium on Foundations of Computer Science, Syracuse, New York*, pages 169–174, 1981.

[CIL87] Benny Chor, Amos Israeli, and Ming Li. On processor coordination using asynchronous hardware. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 86–97, 1987.

[ES92] Ömer Eğecioğlu and Ambuj K. Singh. Naming symmetric processes using shared variables. Unpublished manuscript, 1992.

[Fel68] W. Feller. *An Introduction to Probability Theory and its Applications*, volume 1. Wiley, 3rd edition, 1968.

[JS85] Ralph E. Johnson and Fred B. Schneider. Symmetry and similarity in distributed systems. In *Proceedings of the 4th ACM Symp. on Principles of Distributed Computing*, pages 13–22, 1985.

[Lam86] Leslie Lamport. On interprocess communication (part II). *Distributed Computing*, 1(2):86–101, 1986.

[Lei91] Tom Leighton. *Introduction to Parallel Algorithms and Architectures.* Morgan-Kaufman, 1991.

[LP90] Richard J. Lipton and Arvin Park. The processor identity problem. *Info. Proc. Lett.*, 36:91–94, October 1990.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[Tan81] Andrew Tannenbaum. *Computer Networks.* Prentice Hall, 1981.

[Ten90] Shang-Hua Teng. The processor identity problem. *Info. Proc. Lett.*, 34:147–154, April 1990.