How and When to Be Unique

EXTENDED ABSTRACT*

Shay Kutten† Rafail Ostrovsky‡

Boaz Patt-Shamir§

TR-93-074

November, 1993

Abstract

One of the fundamental problems in distributed computing is how identical processors 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 protocol that solves the problem and terminates with probability 1. 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 no protocol can terminate with probability 1 if n is unknown, and that no finite-state protocol can terminate with probability 1 if the schedule is non-oblivious (i.e., may depend on the history of the shared variable).

We also discuss the dynamic setting (where processes may join and leave the system dynamically), and give a deterministic protocol for the read-modify-write model that needs only 3 shared bits.

---

* Preliminary Version Appeared in Prooceedings of the second Israel Symposium on Theory of Computing and Systems (ISTCS93) Netanya, Israel, June 7th-9th, 1993.

†IBM T.J. Watson Research Center.

‡UC Berkeley. Research partly done during a visit to IBM T.J. Watson Research Center.

§MIT. Research partly done during a visit to IBM T.J. Watson Research Center.

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. This problem, of giving distinct names to identical processes, 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 many applications for some of the most basic distributed tasks, including mutual exclusion, choice coordination, and resource allocation.

On an abstract level, the heart of the problem is "guaranteed" symmetry breaking. Note that most message passing models provide some asymmetry simply by virtue of the topology of the underlying interconnection network. In this paper, we focus only on the completely symmetric shared memory systems. More specifically, we are concerned with the *read-write registers* model [Lam86], where in a single action, a processor can either read or write the contents of a single register. (This model is also known by the name of "atomic registers".) We shall contrast this model with 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. (One of the interesting consequences of this work is sharpening the difference between the computational power of the randomized variants of these two models.)

The formal specification of the problem is as follows. We have a system with n processes, and for each process there is some designated 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.

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

Notice that we insist that processes never err. This rules out trivial solutions that require no communication (e.g., each process chooses a random ID, and the error probability is controlled by the size of the ID space). The original formulation of the problem by Lipton and Park [LP90] had 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 $\{1, \ldots, n\}$.

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 for the dirty memory model was first defined by Lipton and Park in [LP90], where they also give a solution that uses $O(Ln^2)$ shared bits (for any given parameter

$L \geq 0$) and terminates in $O(Ln^2)$ time with probability $1 - c^L$, for some constant $c < 1$. However, if the protocol fails, then it never terminates. Since this event occurs with positive probability, that protocol has infinite expected termination time. The protocol of [LP90] was subsequently improved by Teng [Ten90], who shows how to get running time of $O(n \log^2 n)$ time with probability $1 - 1/n^c$, for any constant $c > 0$. As in [LP90], however, in the event of failure, the protocol never terminates. Recently, Eğecioğlu and Singh [ES92] have obtained independently a randomized protocol that assigns distinct IDs to the processes in $O(n^7)$ expected time using $O(n^4)$ shared variables.

Our Results and Organization of the Paper. In this work, we give tight positive and negative results describing the conditions under which the Processor Identity Problem can be solved. First, we show how to solve the problem: in Section 3 we give a randomized protocol for PIP in the read-write registers model that terminates in $O(\log n)$ expected time, using $O(n)$ shared memory space. The protocol works under the assumption that n is known, and that the schedule is *oblivious* (i.e., the order in which processes take steps does not depend on the actual execution). 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 $\{1, \ldots, n\}$. 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 for the protocol of Section 3 are actually *necessary* to obtain protocols that terminate with probability 1. Specifically, in Section 4 we prove the following results. First, we prove that there is no 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. These results are fairly straightforward. Our last impossibility result seems to be the most interesting. We show that even if n is known, there is no finite-state protocol that solves PIP when the adversary is allowed to be *adaptive*, i.e., when the 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 the results is that 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. 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 access 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 mainly concerned with the *read-write* model, where every step of a process is either a read or a write of a shared variable, followed by some local computation. We shall also discuss *read-modify-write* model, where a process can, in a single indivisible step, access a shared variable, obtain its value, and rewrite it as a function of the process local state and the value of the variable. Precise definitions for these models are given in Appendix A.

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 each process takes at least one step (either a read or a write) every one 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. We also call a time fragment in which all processes are scheduled at least once a *round*.

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 also discuss the dynamic model, where processes may join and leave the system arbitrarily. We explain how a simple variant of our basic protocol can be used for that model. We remark that our protocols rely 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 Terminating Protocol for PIP

We begin with an intuitive description of the protocol. Throughout 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 have unique names.

The protocol is based on a few ideas, designed to implement collision detection, termination detection, and operation with initially dirty memory.

The collision detection is done by 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* of constant size (i.e., a bit). 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 no change is detected, then the process simply proceeds. Otherwise, it concludes that its claimed ID gives rise to a collision. In this case the process chooses a new ID uniformly at random from $\{1, \ldots, N\}$, and then proceeds.

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. 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 above property ensures that we can only underestimate the number of claimed IDs, and thus, if this number is known to be n, then the collision detection can stop. We shall also guarantee that after the set of claimed has stabilized (at size n), then eventually all IDs will be detected as claimed. The number of claimed IDs is counted by the processes in 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" 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 inductively as follows. For each "leaf" register $\mathcal{D}[i, 0]$ we define its *ancestor register* at level j for $j > 0$ by

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

The ancestor relation defines, in the natural way, a directed tree whose root is $\mathcal{D}[1, \log N]$. The semantics of this tree is that each entry $\mathcal{D}[i, level]$ (for $level > 0$) is intended to contain the number of claimed IDs in the range $(i-1)2^{level} + 1 \leq ID \leq i \cdot 2^{level}$. This is done by the following rule. The registers at each level *level* are partitioned into consecutive pairs $\mathcal{D}[2i-1, level]$ and $\mathcal{D}[2i, level]$ (for $0 \leq level \leq \log n$ and $1 \leq i \leq N/2^{level}$). These pairs (whose elements are siblings in the tree) are called *segments*. The idea is that each process maintains its *current level* (initially 0). The process is "responsible" for filling the values

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 (retains current level for summation)*
count : *takes values from* $0..N$*, initially 0 (retains count of claimed IDs in segment)*

Code

```
1   repeat
2     either, with probability 1/2, do
3       if D[tent_ID, 0] ∉ {signature, Λ} then          (read and check for collision)
4         tent_ID ← random({1, ..., N})
5         signature ← Λ
6         level ← 0
7     or, with probability 1/2, do                  (write and record random signature)
8       D[tent_ID, 0] ← signature ← random({0, 1})
9     if level > 0 then                              (summing number of claimed IDs)
10      D [ancestor(tent_ID, level) , level] ← count
11    if level < log N then                    (read the segment in the current level)
12      left ← D [2 · ancestor(tent_ID, level + 1) − 1 , level]
13      right ← D [2 · ancestor(tent_ID, level + 1) , level]
14    if ancestor(tent_ID, level) is the first non-erased register in the segment then
15      count ← right + left                                          (treat Λ as 0)
16      level ← (level + 1) mod ⌈1 + log N⌉
17    else level ← 0                                 (start checking from the leaf again)
18  until D[1, log N] = n                                        (termination predicate)
19  ID ← | {i : D[i, 0] ≠ Λ and i ≤ 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 $\{1, \ldots, n\}$.

for all its ancestors up to that level. It checks the segment associated with its ID at its current level; if the process is the first process in that segment (i.e., either its ancestor register is the only non-erased register in the segment, or else the index of its ancestor is odd), it increments its current level, and writes the sum of the entries in the segment in the ancestor at its new level. Otherwise, it resets it current level to 0, so that 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 $\mathcal{D}[1, \log N] = n$.

As a final twist, since we want the names to constitute the set $\{1, \ldots, n\}$, we perform some kind of "prefix sums". In other words, we compute for each process the *rank* of its ID among the claimed IDs. Using standard techniques (see, e.g., [Lei91]), it can be done in $O(\log n)$ time, given the tree structure already existing in the shared memory.

The formal specification of the protocol is given in Figure 1. To deal with shared memory which is initially dirty, the algorithm uses the following strategy, which we call "read when clean". Each process keeps track of the status of each register of the shared memory. Initially, all registers are marked *dirty*; whenever a write is performed, the written register is marked *clean*. The rule observed by the algorithm is that a process reads only clean registers; this is maintained by erasing the contents of each dirty register with a special value Λ when it is to be read. More specifically, whenever a register needs to be read, its status is checked; if it is *dirty*, then a `write`(Λ) step is inserted before the read. The correctness of the algorithm, despite possible erasures, is maintained since each meaningful value is re-written periodically: at level 0, the collision detection mechanism writes a random signature (different from Λ) every $O(1)$ time units; and for higher levels, each process re-writes the registers under its responsibility in a wrap-around fashion.

Before we turn to analyze the algorithm, let us return to the "read when clean" mechanism. The erasing value (written before a register is ever read) is a special value denoted Λ. Note that by the structure of the algorithm, any register belonging to the tree structure whose leaves correspond to claimed IDs is periodically re-written. Also, the sibling of any such register is erased at some point. For the purpose of addition, Λ is treated as 0.

3.2 Analysis

In this section we sketch the outline of the correctness proof for the protocol. The proofs are presented in Appendix B. We first state the correctness of the protocol.

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.

The following three lemmas show that the termination detection mechanism is correct.

Lemma 1 *If an ID is claimed at some state of the execution, then it remains claimed from that point on.*

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

Lemma 3 *If the protocols reaches a state with n claimed IDs, then in $O(\log n)$ time units all processes exit the main loop.*

The correctness of the collision detection mechanism is expressed in the following pair of lemmas. We remark that the calculations are not optimized. This effects the constant factor in Theorem 1 slightly.

Lemma 4 *Suppose $k > 1$ processes claim the same ID at some state. Then in one time unit, at least $k/16$ of them choose a new tent_ID with probability more than $0.4^{k-1} - N^{-k/2}$.*

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

Theorem 1 is a direct corollary of Lemmas 2, 5 and 3.

To conclude the analysis of the protocol, 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 2). 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 (not necessarily $\{1, \ldots, n\}$).

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 interleaving in it the rank calculation. 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.

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. We remark that this construction is not only dynamic, but it is in fact *self-stabilizing*, i.e., it works even if the initial state of the whole system (shared memory and local states) is arbitrary. We give code for a dynamic protocol in Figure 2 in the appendix.

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. This observation translates fairly easily into a rigorous proof of the time lower bound, and to the proof of necessity of knowledge of n; the proof of impossibility for adaptive adversaries is more difficult.

We analyze the protocols for PIP in terms of Markov chains (we shall use the terminology of [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 the local state of the process 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 process, 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. 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. Since we shall be proving impossibility results, we assume in this section, without loss of generality, that the initial state of the shared memory is fixed and known (this can only add extra power to the protocols). This assumption allows us to have a standard Markov chain, with exactly one node no incoming edges (called hereafter the *source node*), that correspond to the (well defined) initial states of the system.

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

Lemma 6 *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. ∎

We remark that Lemma 6 holds even for infinite state protocols (so long as no zero probability transitions are ever taken).

Using Lemma 6, we can now prove our first necessary condition for PIP.

Theorem 2 *There is no protocol for PIP (including infinite state protocols) that works for unknown n and terminates with probability 1, even for oblivious schedules.*

Proof: For simplicity, suppose $\mathcal{P}$ works for $n = 1$ and $n = 2$. (The argument extends directly to arbitrary different values.) 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 6, 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. ∎

A similar argument, augmented with a standard lower bound technique for parallel computations, shows that any protocol for PIP requires $\Omega(\log n)$ time units.

Theorem 3 *There is no protocol for PIP that terminates in $o(\log n)$ time.*

Proof Sketch: Given an execution of a protocol, we define inductively for each process p_i at any given step k the set of *influencing processes*, denoted $S_i(k)$, by the following rule. At the initial state, the set of influencing processes for process p_i is $S_i(0) = \{p_i\}$ for all i. The influencing set of p_i may grow only when a process reads a register: suppose that p_i reads a register r at step k, and that r was previously written by process p_j at step l. In this case, we define $S_i(k) = S_i(k-1) \cup S_j(l)$. For all other cases, we define $S_i(k) = S_i(k-1)$. Intuitively, the influencing set of a process at any given state is the set of all processes that might have effected the computation of that process up to that point. It is easy to see that in any run of a protocol for PIP, a process may terminate only when its influencing set is $\{1, \ldots, n\}$. For suppose not, i.e., there exists a run ρ of the system, in which p_i terminates at step k with $j \notin S_i(k)$. Then there exists another run ρ' which is identical to ρ from the point of view of p_i, in which p_j advances in lockstep with p_i, maintaining identical local state. Since p_i terminates in ρ' iff it terminates in ρ, and since termination in ρ' violates the uniqueness requirement, we have reached a contradiction, proving that a process p_i may terminate at a step k only if $S_i(k) = \{1, \ldots, n\}$.

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

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.) For such adversaries we have the following theorem.

Theorem 4 *There is no finite state protocol that solves the Processor Identity Problem with probability* 1 *using read-write registers 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 $\mathcal{P}$), 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 2, 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 process into irreducible processes. In graph theoretic language, consider the Markov graph $G_\mathcal{P}$: it is a directed graph; we decompose it into strongly-connected components [Eve79]. 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 7 *Consider a global state* s^* *which corresponds to a state* s *in a terminal component of* $G_\mathcal{P}$*. 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 process* p_i *only,* 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 an irreducible Markov chain, all states are persistent, i.e., with probability 1, s^* eventually occurs again. ∎

Lemma 7 gives rise to the following immediate corollary.

Corollary 8 *Let* s^* *be a global state which corresponds to a state in a terminal component of* $G_\mathcal{P}$*, 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 process* p_i *only,* $\bar{v}$ *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 6, that 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 8) 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 8 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 that solves PIP for any fair adversary.*

The unbounded protocol is presented in Figure 3 in the Appendix.

5 PIP in the Read-Modify-Write Model

In this section we give positive and negative results for PIP in the read-modify-write model. Due to the lack of space, we only state the results here. The proofs are given in Appendix E.

Theorem 6 *In the read-modify-write model, there exists a deterministic protocol for PIP that works with 3 shared bits under any schedule.*

Theorem 7 *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.*

Acknowledgment

The third author would like to thank 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 environment. In *28th Annual Symposium on Foundations of Computer Science, White Plains, New York*, pages 337–346, 1987.

[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.

[Eve79] Shimon Even. *Graph Algorithms.* Computer Science Press, 1979.

[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.

[Spe87] Joel Spencer. *Ten Lectures on the Probabilistic Method.* SIAM, Philadelphia, Pennsylvania, 1987.

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

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

APPENDIX

A Model Definitions

We first make the following abstract definition, to be used in the definition of randomized protocols.

Definition 1 *For a given domain A and range B, a* probabilistic function F of type $A \rightarrow B$ *is a collection of deterministic functions of type $A \rightarrow B$, together with a probability space over the collection. The value of $F(a)$ for $a \in A$ is determined by first picking a function $f \in F$ at random according to the given distribution on F, and then applying f to a.*

We can now define the *read-write* model

Definition 2 *A distributed system satisfies the* atomic registers *condition if each step of each process p_i is one of the following.*

1. $\texttt{read}_i(r_j)$ *action, where r_j is a register, and the next state of the system satisfies the following constraints.*
   - *The next state of the system is unchanged for all registers, and for all processes other than p_i.*
   - *The next state of p_i changes as a probabilistic function of its previous local state and of the state of r_j.*
2. $\texttt{write}_i(r_j)$ *action, where r_j is a register, and the next state of the system satisfies the following constraints.*
   - *The next state of the system is unchanged for all registers other than r_j, and for all processes other than p_i.*
   - *The next state of p_i changes as probabilistic function* only of its previous state. *(This reflects the fact that p_i "knows" that it has made the $\texttt{write}_i$ action).*
   - *The next state of r_j changes as a* function *of the previous state of p_i.*

In the *read-modify-write* model, we have only one type of action, termed `access`.

Definition 3 *A distributed system satisfies the* read-modify-write *condition if for each process p_i and register r_j, the $\texttt{access}_i(r_j)$ action satisfies the following constraints.*

- *The next state of the system is unchanged for all registers other than r_j, and for all processes other than p_i.*
- *The next state of p_i changes as a probabilistic function of its previous local state and of the previous state of r_j.*
- *The next state of r_j changes as a function of the previous state of p_i.*

B Correctness Proof for the Protocol

In this section we give sketches of the proofs for the claims used in the proof of Theorem 1, i.e., the correctness of the algorithm in Figure 1.

We start with a simple fact, that follows from the fact that $N = c \cdot n$.

Lemma 9 *Whenever a process chooses a new value for tent_ID, this value is unique with probability at least* $1 - \frac{1}{c}$.

Proof: Since the choices of *tent_ID* are independent and uniform over $\{1, \ldots, N\}$, and since the number of claimed ID is at most n, we have that the probability to choose a claimed ID is at most $\frac{n}{N} = \frac{1}{c}$. ∎

Lemma 1 *If an ID is claimed at some state of the execution, then it remains claimed from that point on.*

Proof Sketch: The claim is immediate for IDs that are unique for the remainder of the execution. Now suppose that two or more processes claim some ID. The lemma follows from the observation 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. ∎

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

Proof Sketch: By induction on the level numbers. 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 $2i - 1$ and $2i$ are claimed. By lemma 1 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. ∎

Lemma 3 *If the protocols reaches a state with n claimed IDs, then in $O(\log n)$ time units all processes exit the main loop.*

Proof Sketch: We show, by induction on i, that in $O(\log n) + O(i)$ time units, the set of locations accessed by the processes at level i is fixed, and that any read from level i of the tree returns the correct value. The base case, for level 0, is given by the assumption The "read when clean" rule will have no effect on level 0 from that point on, since the locations read by processes at that level will remain fixed. Next, notice that in a time interval of $O(\log n)$ units, for each process there is a state with $level = 0$. Now, to prove the inductive step it is sufficient to show that no erasures occur at level $i + 1$. This follows from the fact that the reads from level i are correct, and hence only one process will write at level $i + 1$. ∎

Lemma 4 *Suppose $k > 1$ processes claim the same ID at some state. Then in $O(1)$ time units, at least $k/16$ of them choose a new tent_ID with probability more than $0.4^{k-1} - N^{-k/2}$.*

Proof Sketch: If one of the processes has terminated, then by Lemma 2 we are done. So suppose that none of the processes has terminated. Consider an execution fragment where each of the k processes accesses the claimed register at least twice. The duration of any such fragment is $O(1)$ time units. Denote the sequence of accesses to the claimed variable in this time interval by $a_1, a_2, \ldots, a_m$. Denote the second access of process p_j by a_{i_j}, for

$1 \leq j \leq k$. Note that by the "read when clean" rule, the second access is never `write`(Λ). Now 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 second 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 probability $1/8$, for $1 \leq j \leq \lfloor k/2 \rfloor$. This is because the previous access to the register was by another process, that wrote (with probability $1/2$) a value that differs from $tent\_ID_i$ (with probability $1/2$), and p_i makes a read (with probability $1/2$). 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 $1/8$. 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 the generalized Chernoff Bound [Spe87]. We need to subtract the probability that the $a_{i_{2j}}$ are `write`(Λ) events, which is $N^{-k/2}$. ∎

Lemma 5 *After $O(\log n)$ expected time units, the number of claimed IDs is n.*
Proof Sketch: Suppose that the number of collisions at some state is b. Using Lemma 4 and Lemma 9, it can be shown that in $O(\log n)$ expected time, the maximal number of colliding processes per ID is at most 31. Using the same argument as in Lemma 4, it is easy to show that in additional $O(\log n)$ expected time, all IDs are unique. ∎

C Dynamic Protocol for PIP

In the code given in Figure 2, we use a slightly different randomization technique: instead of flipping a coin whether to read or write (as in the algorithm of Figure 1, we associate with each ID two registers, and we choose randomly from which to read or write.

Shared Variables

$\mathcal{D}$	: *a vector of N pairs of bits*	

Local Variables

signature	: *a vector of two bits*	*(retains state of claimed segment)*
ID	: *output value*	*(to stabilize on a unique value)*
j, pos	: *temporary indices*	

Code

```
1  repeat forever
2    pos ← random({1,2})
3    if signature[pos] ≠ D[ID,pos] then        (read location, check for collision)
4      ID ← random({1,...,N})
5      for all j ∈ {1,2} do signature[j] ← D[ID,j]
6    else                                      (ID seems unique)
7      D[ID,pos] ← signature[pos] ← random({0,1})   (write new random signature)
```

Figure 2: Dynamic algorithm for PIP using $2N$ shared bits.

The algorithm in Figure 2 does not need to guarantee that a claimed ID will always remain claimed. It is sufficient to guarantee that any collision will be detected in expected constant time. We therefore use only spatial randomization here, that controls the location the process accesses next. The correctness of the algorithm is stated in the theorem below.

Theorem 8 *The algorithm in Figure 2 stabilizes in $O(\log n)$ expected time units with all IDs unique.*

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

D Unbounded Memory Algorithm for PIP

Shared Variables

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

Local Variables

ID : *output value, initially* $random(\{1,\ldots,N\})$

signature : *a sequence of bits, initially empty sequence (retains state of claimed segment)*

Code

```
1 repeat
2    either, with probability 1/2 do
3            if signature ≠ D[ID] then                          (read and check)
4               ID ← random({1,...,N})
5               signature ← 0
6    or, with probability 1/2, do                    (append a new random signature)
7            D[ID] ← signature ← (2 · signature + random({0,1}))
8 until |{i : D[i] ≠ 0}| = n                                   (termination predicate)
9 ID ← |{i : D[i] ≠ empty sequence and i ≤ tent_ID}|   (output value: rank of tent_ID)
```

Figure 3: Unbounded space algorithm for PIP under any adversary

Theorem 9 *The algorithm in Figure 3 stabilizes with probability* 1 *in expected* $O(\log n)$ *time units.*

Proof Sketch: Intuitively, the idea in the algorithm is that each process maintains a complete history of the location it claims. The main step in proving the theorem, is showing that every collision is eventually detected. This follows from the fact with probability 1, one process eventually reads what another process wrote, and since that the probability that two colliding processes have the same history forever is 0. ∎

E PIP in 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 2 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.

E.1 Solving PIP with 3 Shared Bits

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 process, by the read-modify-write assumption. Also, we assume that the shared variable is initialized with a special initial 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 given in Figure 4. We summarize in the following theorem.

Theorem 6 *In the read-modify-write model, there exists a deterministic protocol for PIP that works with 3 shared bits under any schedule.*

E.2 Impossibility for Self-Stabilizing Protocols

In this section we describe briefly how can the technique of Theorem 4 be extended to the read-modify-write model, under the stronger assumption that the initial state is arbitrary.

Theorem 7 *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.*

Proof Sketch: As in Theorem 4, we prove the theorem by showing an adaptive adversary under which, with some fixed probability ϵ_0, the protocol cannot stabilize in any given time T.

We again consider the Markov graph, and its decomposition into strongly connected components. Now, by the self-stabilization assumption, we can let the first state of the

system be such that the two processes are in identical states in some reachable terminal component. We now apply the schedule in which p_1 takes steps until the values of the shared memory vector recur (since Lemma 7 and Corollary 8 hold in this case also), and then let p_2 take steps until the shared memory state recurs, and so on for T time steps. This yields (with some probability ϵ_0) a fair schedule in which the view of p_1 and p_2 is identical to the view of them running alone, and hence they either don't stabilize in time T (i.e., keep changing their IDs), or else the initial collision of IDs persists forever. ∎

Shared Variable

message: takes values from {`init`, `ready`, `accept`, 0, 1, `ack`, `end`}, initially `init`

Local Variables

ID : output value

mode : takes values from {*start*, *get*, *seek*, *give*, *done*}, initially *start*

rem : integer

Code

```
repeat
  case mode
    start: if message = init then
               mode ← seek
               ID ← 0
               message ← ready
           else if message = ready then
               mode ← get
               ID ← 0
               message ← accept
           else message ← message
    seek:  if message = accept then
               mode ← give
               message ← ID mod 2
               rem ← ⌊ID/2⌋
           else message ← message
    get:   if message ∈ {0, 1} then
               ID ← 2 · ID + message
               message ← ack
           else if message = end then
               ID ← ID + 1
               mode ← seek
               message ← ready
           else message ← message
    give:  if message = ack then
               if rem ≠ 0 then
                   message ← rem mod 2
                   rem ← ⌊rem/2⌋
               else mode ← done
                   message ← end
           else message ← message
  end case
until mode = done
```

Figure 4: Deterministic algorithm for PIP in the read-modify-write model, using 3 shared bits