Randomized Naming Using Wait-Free Shared Variables*

Alessandro Panconesi[†] Freie Universität Berlin

Marina Papatriantafilou[‡] Max Planck Institut

Philippas Tsigas[§] Max Planck Institut

Paul Vitányi[¶] CWI & Universiteit van Amsterdam

Abstract

A naming protocol assigns unique names (keys) to every process out of a set of communicating processes. We construct a randomized wait-free naming protocol using wait-free atomic read/write registers (shared variables) as process intercommunication primitives. Each process has its own private register and can read all others. The addresses/names each one uses for the others are possibly different: Processes p and q address the register of process r in a way not known to each other. For n processes and $\epsilon > 0$, the protocol uses a name space of size $(1+\epsilon)n$ and $O(n \log n \log\log n)$ running time (read/writes to shared bits) with probability at least $1-o(1)$, and $O(n \log^2 n)$ overall expected running time. The protocol is based on the wait-free implementation of a novel α-*Test&SetOnce* object that randomly and fast selects a winner from a set of q contenders with probability at least α in the face of the strongest possible adaptive adversary.

Keywords: Naming problem, Symmetry breaking, Unique process ID, Asynchronous distributed protocols, Fault-tolerance, Shared memory, Wait-free read/write registers, Atomicity, Test-and-set objects, Randomized algorithms, Adaptive adversary.

---

*This work was performed during a stay of AP, MP, and PT at CWI, Amsterdam.

[†]Supported by an ERCIM Fellowship. Address: Freie Universität Berlin, Institut für Informatik, Takustr. 9, 14195 Berlin, Germany; Email: ale@inf.fu-berlin.de)

[‡]Partially supported by a NUFFIC Fellowship and the European Union through ALCOM ESPRIT Project Nr. 7141. Address: Max Planck Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany; Email: ptrianta@mpi-sb.mpg.de)

[§]Partially supported by NWO through NFI Project ALADDIN under contract number NF 62-376 and by the European Union through ALCOM ESPRIT Project Nr. 7141. Address: Max Planck Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany; Email: tsigas@mpi-sb.mpg.de)

[¶]Partially supported by NWO through NFI Project ALADDIN under contract number NF 62-376 and by the European Union through NeuroCOLT ESPRIT Working Group Nr. 8556. Address: CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands; Email: paulv@cwi.nl)

1 Introduction

A *naming* protocol concurrently executed by each process out of a subset of n processes selects for the host process a *unique* name from a common name space. The name space should be small, preferably of size n. The processes may or may not have a name to start with. If they do, the resulting variant of the naming problem is called the *renaming* problem.

In a distributed or concurrent system, distinct names are useful and sometimes mandatory in a variety of situations including mutual exclusion, resource allocation, leader election and choice coordination. In such cases a naming protocol can be put to good use. When processes are created and terminated dynamically—a common occurrence in distributed and concurrent systems—the name space may grow while the number of processes remains bounded. A renaming procedure is used to size down the name space. Examples of network protocols that crash on duplicate names or perform more efficiently for small name ranges are found in [23] and [26]. A naming protocol is also useful in allocation of identical resources with a name as a permit to a resource. Since our algorithms are wait-free (see below) they are also highly fault-tolerant. Managing the assignment of resources to competing processes corresponds to a repetitive variant of the naming problem [6]. In the sequel we also write "key" for "name" and "key range" for "name space."

Interprocess Communication: We use interprocess communication through shared memory and allow arbitrarily initialized shared memory (dirty memory model) as in [19]. Shared memory primitives such as wait-free atomic read/write registers [17, 18] are widely used in the theory of distributed algorithms [12]. A deterministic protocol executed by n processes is *wait-free* if there is a finite function f such that every non-faulty process terminates its protocol executing a number $\leq f(n)$ of steps regardless of the other processes execution speeds (or crash failures). In other words, a wait-free solution is $(n-1)$-resilient to process crash failures. A *randomized* protocol is wait-free if $f(n)$ upper bounds the *expectation* of the number of steps, where the expectation is taken over all randomized system executions against the worst-case adversary in the class of adversaries considered (in our results the adaptive adversaries). Our constructions below use single-writer multi-reader wait-free atomic registers as constructed in [25, 18] and used in [7, 8, 15]. We also write "shared variable" for "register."

Anonymous Communication Model: Every register can be written by exactly one process and can be read by all other processes—this way the writing process can send messages to the other processes. If the processes use a common index scheme for other processes registers (an initial consistent numbering among the processes as it is called in [7, 8, 15]), then optimal naming is trivial by having every process rank its own number among the other values and choose that rank-number as its key. To make the problem nontrivial, every process has its own private register and can read all other registers but the processes use possibly different index schemes. That is, processes p and q each address a register owned by process r in a possibly different way not known to each other. This may happen in large dynamically changing systems where the consistency requirement is difficult or impossible to maintain [19] or in cryptographical systems where consistency is to be avoided. In this model we cannot use the consensus protocols of [4, 5, 14] or the test-and-set implementation outlined in [2].

Symmetric Shared Memory: Another way to prevent trivial ranking is using the symmetric shared memory model. A shared memory is *symmetric* if it consists of a set of identical processes communicating through a pool of shared variables each of which is read and written by all processes [16, 10].

Complexity Measures: The computational complexity of distributed deterministic algorithms using shared memory is commonly expressed in number and type of intercommunication primitives required and the maximum number of sequential read/writes by any single process in a system execution. Local computation is usually ignored. We use wait-free atomic read/write registers as primitives. Such primitives must be ultimately implemented wait-free from single-reader single-writer wait-free atomic read/write bits (that in turn are implementable from mathematical versions of hardware "flip-flops" [17]). The most efficient such implementations use [18] to reduce a multiuser multivalue register to single-reader single-writer multivalue registers, and [17] to reduce the latter to single-reader single-writer bits. To standardize complexity and to make comparisons to other algorithms unambiguous we express time- and space complexity in terms of read/writes to the elementary shared bits.

Randomization: The algorithms executed by each process are randomized by having the process flip coins (access a random number generator). In our randomized algorithms the answers are always correct—each process always gets a unique key—but with small probability the protocol takes a long time to finish. We use the customary assumption that the coin flip and subsequent write to shared memory are separate atomic actions. To express the computational complexity of our algorithms we use (i) the worst-case complexity with probability $1 - o(1)$, or (ii) the expected complexity, over all system executions and with respect to the randomization by the processes and the worst-case scheduling strategy of an adaptive adversary.

Previous Work: The agreement problem in the deterministic model of computation (shared memory or message passing) is unsolvable in the presence of faults [11, 13, 20]. Surprisingly, [6] showed that the *renaming* problem, which requires a nontrivial form of interprocess "agreement," is solvable in the message passing model. Their solution is t-resilient—up to t crash failures are tolerated—and uses a name space of size only $n+t$, but it takes exponential time (in n). This protocol was later transformed in [7] to two solutions for the asynchronous, shared memory model that are wait-free and achieve running times of $(n+1)4^n$ and n^2+1 and key range sizes of $2n-1$ and $(n^2+n)/2$, respectively. Recently, [9] demonstrated a wait-free long-lived shared memory implementation for renaming k out of n processes, using $O(k^3)$ steps and achieving key range size $O(k^2)$. For a deterministic, wait-free solution in the asynchronous, shared memory model a key range of size $2n-1$ is necessary [15].

The above results use asymmetric shared memory in the form of single-writer multi-reader wait-free atomic read/write registers and each "step" reads or writes one such register. Moreover, the global address of each register is known to all.

There is a plethora of other work on the naming problem using shared memory cited in [19, 16, 10]. We discuss what is relevant to this paper. In [16] it is shown that using *bounded* symmetric shared memory, both a deterministic solution and a randomized wait-free solution against an *adaptive* adversary (Appendix A) are impossible. They give a wait-free randomized *unbounded* symmetric shared memory solution against a fair adaptive adversary with a key range of size n and a logarithmic expected number of *rounds*—time units during which every process makes at least one step—assuming $\log n$-bit registers. They show that unbounded memory is also necessary against an adaptive adversary. They also give an $O(\log n)$ expected time solution with key range size n against a fair *oblivious* adversary using $O(n)$ shared memory consisting of $\log n$-bit registers. Independently, [10] gave a randomized solution in the bounded symmetric memory model with key range size n running in expected $O(n^6)$ time against a fair oblivious adversary using $O(n^4)$ shared atomic multiwriter, multireader bits.

To summarize: For asynchronous bounded shared memory, in the *asymmetric* case deterministic wait-free solutions are expensive (in terms of achievable key range) and in the *symmetric* case both deterministic wait-free solutions and randomized wait-free solutions assuming an adaptive adversary are impossible. The remaining case is treated below.

Present Results: We show that randomization can yield wait-free, inexpensive solutions in terms of time, space, and key range for asynchronous asymmetric bounded shared memory models (single-writer multi-reader wait-free atomic shared registers). We use the anonymous communication model were processes have no initial names (equivalently, have no global consistent indexing of the other processors). Our construction requires several algorithms that in the end give a randomized wait-free naming algorithm. We assume the adaptive adversary (the strongest adversary).

Our first algorithm is the implementation of an α-*Test&SetOnce* object: a one-shot test-and-set that guarantees that among the $q \leq n$ competing processes invoking it, there will be a unique winner with probability α, where α is a parameter that can be chosen arbitrarily close to 1. The object is safe; that is, at most one process can be a winner. When invoked by q (out of n) processes, it uses $O(\log q)$ 1-writer n-reader shared bits per process. The running time is $O(n \log q)$ read/writes on such bits. These properties are shown in Theorem 1. In our applications we typically have $q = O(\log n)$ with high probability. Using more complex primitives, the object can be implemented by: (a) using n copies of 1-writer n-reader $O(\log \log q)$-bit read/write registers with running time of $O(n \log q)$ read/writes on these registers, or (b) a single n-writer n-reader 1-writer-per-component n-component composite register with $\log \log n$-bit components (snapshot memory [3, 1]) with running time $O(\log q)$ read/writes per process on the composite register.

The second algorithm is a wait-free naming algorithm, SEGMENT , using α-*Test&SetOnce* objects. Given any $\epsilon > 0$, SEGMENT uses a name space of size $(1+\epsilon)n$, where n is the number of processes. The protocol is always correct in the sense that all non-faulty processes receive distinct names. The running time is a random variable whose value is $O(n \log n \log \log n)$ bit operations with high probability. By "high probability" we mean that the probability that the running time exceeds the above value is $o(1)$, a quantity that tends to 0 as n grows, Lemma 3. In fact, we prove that the *maximum running time* among all non-crashing processes is $O(n \log n \log \log n)$ bit operations with probability $1 - o(1)$. It is still possible that the expectation of the running time over all coin flip sequences is infinite. Our next Theorem 2 shows that with a minor modification a proof similar to that of Lemma 3 demonstrates that the expected running time of the modified protocols of the involved processes can be bounded by $O(n \log^2 n)$ and hence SEGMENT is "wait-free."

The paper is organized as follows. Section 2 and Appendix A spell out our assumptions and our model of computation. Appendix B shows that the simple approach doesn't work and motivates the introduction of the α-*Test&SetOnce* object in Section 3. This object is used in Section 4 to obtain the naming protocol.

2 Preliminaries

Processes are sequentially executed finite programs with bounded local variables communicating through single-writer, multi-reader bounded wait-free atomic registers (shared variables). The latter are a common model for interprocess communication through shared memory as discussed briefly in Section 1. For details see [17, 18] and for use and motivation in distributed protocols see [7, 8, 15].

2.1 Shared Registers, Anonymous Communication, Atomicity

Every read/write register is *owned* by one process. Only the owner of a register can write it, while all the other processes can read it. In one *step* a process can either: (i) read the value of a register, (ii) write a value to one of its own registers, or (iii) flip a local coin (invoke a random number generator that returns a random bit), followed by some local computation. The communication is *anonymous*: While each process has its own private register and can read all others, the addresses/names each one uses for the others are possibly different: Processes p and q each address the register of process r in a way not known to each other.

We require the system to be *atomic*: every step of a process can be thought to take place in an indivisible instance of time and in every indivisible time instance at most one step by one process is executed. The atomicity requirement induces in each actual system execution total orders on the set of all of the steps by the different processes, on the set of steps of every individual process, and on the set of read/write operations executed on each individual register. The *state* of the system gives for each process: the contents of the program counter, the contents of the local variables, and the contents of the owned shared registers. Since processes execute sequential programs, in each state every process has at most a single step to be executed next. Such steps are *enabled* in that state. There is an *adversary* scheduling demon that in each state decides which enabled step is executed next, and thus determines the sequence of steps of the system execution. There are two main types of adversaries: the *oblivious* adversary that uses a fixed schedule independent of the system execution, and the much stronger *adaptive* adversary that dynamically adapts the schedule based on the past initial segment of the system execution. Our results hold against the adaptive adversary—the strongest adversary possible.

The computational complexity of a randomized distributed algorithm in an adversarial setting and the corresponding notion of wait-freeness require careful definitions. To not distract the reader we delegated the rigorous novel formulation of adversaries as restricted measures over the set of system executions to the Appendix A. We believe it is interesting in its own right and will be useful elsewhere. For now we assume that the notions of system execution, wait-freeness, adaptive adversary, and expected complexity are familiar. A randomized distributed algorithm is *wait-free* if the expected number of read/writes to shared memory by every participating process is bounded by a finite function $f(n)$, where n is the number of processes. The expectation is taken over the probability measure over all randomized system executions against the worst-case adaptive adversary.

2.2 Obvious Strategy Doesn't Work

In Appendix B we analyze a naming strategy that first comes to mind and show it doesn't work out in the sense that $f(n)$ bounding the expected number of read/writes to shared memory is at least exponential. Namely, as soon as there is more than one process claiming a key the adversary can make all such processes fail. This problem can be resolved by a test-and-set mechanism that ensures a winner among a set of claiming processes. However, existing constructions such as [2] require all processes to have a consistent numbering—the model is not anonymous. As pointed out in the introduction, this would render the naming problem trivial: just rank the numbering and choose your rank as a key, see also [7, 8, 15]. To resolve this problem we introduce a probabilistic α-*Test&SetOnce* object that selects a winner with high probability and doesn't require a consistent initial numbering among the processes.

3 Probabilistic α-*Test&SetOnce* Object

An α-*Test&SetOnce* object shared by n processes is a probabilistic, wait-free object with the following functionality: For every $0 \leq \alpha < 1$ we can construct the object such that when it is concurrently invoked by any subset of the user processes it selects a winner with probability $\geq \alpha$. If there are $q \leq n$ processes competing for the same object, then the maximum number of shared bit accesses performed by any process has expectation $O(n \log q)$. Typically, $q := O(\log n)$ so that the expectation is $O(n \log \log n)$.

The object is based on the following property of the geometric distribution. Suppose there are q random variables X_i identically and geometrically distributed, $\Pr[X_i = k] = (1-p)p^k$. Then, with "good probability" there will be a unique maximum, $X_i > X_j$ for some i and all

$j \neq i$.

3.1 Synchronous Algorithm

Consider n processes numbered 1 through n and an $n \times \infty$ matrix $A = (a_{i,j})$ $(1 \leq i \leq \infty, 1 \leq j \leq n)$. Processes $p_1, p_2, \ldots, p_q$ $(q \leq n)$ enter the competition which is expressed by initially setting $a_{1,p_i} := 1$ for $1 \leq i \leq q$ and filling the remainder of A with zero entries. The game is divided into rounds $k := 1, 2, \ldots$. In each round, every process that is still in the game independently flips an identical coin with probability s of success and $s - 1$ of failure. In each round k, every process p_i with $a_{k,p_i} = 1$ flips its coin. If p_i's coin flip is successful then it *steps forward* (sets $a_{k+1,p_i} := 1$) else it *backs-off* (resets $a_{l,p_i} := 0$ $(1 \leq l \leq k)$ and exits). In each round there are three mutually-exclusive possible outcomes: (i) exactly one process steps forward and all others back-off—the process is declared the *winner* and the game ends; (ii) all processes back-off, in which case the game ends with no winner; (iii) more than one process steps forward, in which case the game continues, until one of the two aforementioned events occurs.

Let $f(q)$ denote the probability that the game ends with a winner for an initial number q of competing processes. The exact behavior of $f(q)$ seems hard to analyze. Fortunately, the next lemma gives an easy proof of a statement that is good enough for our purposes. We define $f(0) = 0$ and $f(1) = 1$.

Lemma 1 *Let $0 < s < 1$. Then*

(i) *$f(2) = 2s/(1 + s)$ and*

(ii) *for all $q \geq 2$, $f(q) \geq f(2)$.*

Proof. Suppose that after the first coin flip, k out of q initial processes step forward. Since the number of rows available for the game is unbounded, the probability of having a winner at this point is exactly $f(k)$. Let X be a random variable denoting the number of processes that step forward. Then, the probability of the game ending with a winner is:

$$f(q) = \sum_{k=0}^{q} f(k) \Pr[X = k]$$

Recalling that $f(0) = 0$ and $f(1) = 1$, this equation can be rewritten as:

$$f(q) = \Pr[X = 1] + f(q) \Pr[X = q] + \sum_{k=2}^{q-1} f(k) \Pr[X = k] \tag{1}$$

The probability of having exactly k out of q processes stepping forward to the next row is given by

$$\Pr[X = k] = \binom{q}{k} s^k (1 - s)^{q-k} \tag{2}$$

For the case $q = 2$, these two equations give

$$f(2) = s^2 f(2) + 2s(1 - s) \tag{3}$$

which implies the first part of the lemma.

We prove the second part of the lemma by induction. The base case, $f(2) \geq f(2)$, is trivial. For the inductive step, assume that $f(k) \geq f(2)$ for all $2 \leq k < q$. Using (1) and (2), it follows from the induction hypothesis that

$$\begin{aligned}
f(q) &= \frac{\Pr[X=1] + \sum_{k=2}^{q-1} f(k)\Pr[X=k]}{1 - \Pr[X=q]} \\
&\geq \frac{\Pr[X=1] + f(2)\sum_{k=2}^{q-1} \Pr[X=k]}{1 - \Pr[X=q]} \\
&= \frac{\Pr[X=1] + f(2)(1 - \Pr[X=0] - \Pr[X=1] - \Pr[X=q])}{1 - \Pr[X=q]} \\
&\geq f(2).
\end{aligned}$$

The last inequality is equivalent to

$$\Pr[X=1] - f(2)(\Pr[X=0] + \Pr[X=1]) \geq 0$$

which can be verified using (2) and (3). □

The next lemma shows that, with high probability, the game ends very quickly with a winner.

Lemma 2 *Let $1 \leq q \leq n$. Then, the probability that there is a winner within r rows is at least $f(2) - ns^r$.*

Proof. Let W_r be the event that there is a winner within r rows. Then for $q = 1$ $\Pr[W_r] = 1$. For other values of q

$$\begin{aligned}
\Pr[W_r] &= \Pr[\text{there is a winner}] - \Pr[\text{there is a winner after } r \text{ rows}] \\
&\geq f(q) - \Pr[\text{some process makes it for at least } r \text{ rows}] \\
&\geq f(2) - qs^r \\
&\geq f(2) - ns^r.
\end{aligned}$$

□

An important corollary of this lemma is that, by choosing s and r appropriately, the probability of having a winner within $O(\log q)$ rows can be set arbitrarily close to 1. In other words, infinitely many rows are not needed (but simplify the analysis). If we want a probability of success of $\alpha = 1 - \epsilon$ we need to satisfy

$$\frac{2s}{1+s} - ns^r \geq 1 - \epsilon.$$

By setting $s = 1 - \epsilon/2$ the number of rows needed would be only $r \approx (2/\epsilon)\log(n/\epsilon)$. For instance, for $\alpha = .9$ we would need $r \approx 20(\log n + 4)$ and for $\alpha = .99$ we would need $r \approx 200(\log n + 7)$.

3.2 Asynchronous Implementation

Let the entries of matrix A correspond to the states of 1-writer n-reader bits and let there be only r rows, so A is an $r \times n$ matrix. The j-th bit of each array can be written only by process j but can be read by all processes.

Definition 1 When a process steps forward from row $k-1$ to row k it first sets its private bit at row k to 1 and then reads the other bits of row k. If they are all 0 the process is said to be *lucky at row k*.

Even though in an asynchronous system a process cannot determine whether it reached a certain row alone or whether slower processes will eventually reach the same row, it suffices that it can determine whether it is lucky: the geometric distribution ensures that a process that has not backed-off after many rows (say $k \approx \log n$) is, with high probability, the only one left. Trivially, to be lucky at row k ($1 \leq k \leq r$) is necessary to be a winner and, as we will show, to be lucky at row r is sufficient to be a winner.

Theorem 1 *For every $0 < s < 1$, the α-*TASONCE *protocol implements a α-*Test&SetOnce *object that selects a unique winner among a set of invoking processes with probability at least $\alpha := 2s/(1+s) - o(1)$ (provided no processes crash) and never selects more than one winner. The object can be invoked repeatedly until the key is assigned, provided no crashes occur. If the object is invoked by q out of n processes the invocation uses $O(n \log q)$ 1-writer n-reader atomic single bit registers and has worst case running time of $O(n \log q)$ read/writes of the shared bits.*

Proof. In Figure 1 every process p owns an array of atomic bits denoted by $a[1..r, p]$—one bit for each row of the game. The i-th row $a[i, 1..n]$ has one bit $a[i, p]$ owned by process p ($1 \leq p \leq n$). Initially, in line 1 of Figure 1, the process checks whether the object is currently occupied by *other* processes trying to grab the key or whether initial memory is dirty (possibly by a previous competition). If so, then it exits reporting a failure by way of line 9. Line 9 resets all bits $a[i, p]$ ($1 \leq i \leq r$) owned by the process to 0 to clean its "own" possibly dirty memory for future tries. (Of course, this needs to be done only once at the start of each process and we can add a line of code to this effect.) Lines 2 through 8 implement the following algorithm: Determine if you are lucky at the current row. If yes, then step forward with probability 1, otherwise with probability s. The value of s is the same for all processes. A process *wins* if it is lucky at row r, otherwise it *fails*. We will show below that at most one process can win and hence that the protocol is safe. Before exiting by reporting a failure, the protocol "cleans up" its private bit array (line 9). This is done to make the object reusable if no process wins and no crashes occur so that eventually every non faulty process gets a name.

From a probabilistic point of view it is immaterial whether the coins are flipped synchronously or asynchronously. Because the coin flips are independent, the rate at which processes back off remains essentially unchanged which is the key to the probabilistic analysis of the asynchronous process. Another main ingredient in the proof is a simple upper bound on the number of lucky processes per row. Notice also that if a process is actually the winner at some row—all other processes backed off—from then on it will step from one row to the next with probability 1.

The relevant properties of the protocol are: Liveness: every non faulty process executes the protocol for a bounded number of steps, regardless of other processes speeds or crash failures; safety: at most one process wins; and, if the number of rows is $O(\log q)$ then the probability that among $q \leq n$ competing processes there is a winner is α—a parameter that can be set arbitrarily close to 1. [1]

Claim 1 (Liveness) *α-TASONCE is wait-free and uses at most $2n(r+1)+r$ read/writes to 1-writer, n-reader shared bits.*

Proof. A process p invoking the protocol either backs off immediately, executing at most $2n+r$ steps, or joins the competitition. Then, it either will win executing at most $2n(r+1)$ steps or back off and lose executing at most $2n(r+1)+r$ steps. □

In the remainder of the section we consider a system of $q \leq n$ processes executing the protocol of Figure 1, and executions for this system such that no crashes occur and show that in this case a process gets a key—and hence it is captured—with probability α.

Claim 2 *Let B be the set of processes that back off during an execution and let $b := |B|$. For each row, the number of lucky processes is at most $b+1$ and at most one of them is outside B.*

Proof. A process which does not exit right away after executing line 1 is called a *competing* process. For every row *row* ($1 \leq row \leq r$), every still competing process p first sets $a[row, p] := 1$ by executing $WRITE(a[row,p],1)$ in line 3 of the protocol, and subsequently reads the other bits in the row by executing the loop of line 4. Suppose by way of contradiction that two process p and p' do not back off and both are lucky at row *row*. We can assume that p executes its $WRITE(a[row,p],1)$ before p' executes its $WRITE(a[row,p'],1)$. Since p and p' are not backing off, these bits will stay 1. But the order of atomic events is

$$WRITE(a[row,p],1) < WRITE(a[row,p'],1) < READ(a[row,p'])$$

which contradicts that p' is lucky because $a[row,p]=1$ by the time p' reads it. □

Consequently, among the processes that do not back off, at most one can be lucky at a certain row. As for the processes which do back off, all of them could be lucky. Consider for

[1] In the case of crashes we need not bother to estimate the probability. This is because the adversary is forced to "sacrifice" processes: for every invokation either some process crashes or one process wins the game with probability $\alpha \approx 1$. Given enough objects, all non-faulty processes will sooner or later get a key. The problem, discussed below, is how to make this happen fast for all processes using as few objects as possible.

```
                                                                   {Shared Declarations}
param  r: int ;                                      {number of rows for the game = O(log q}
param  s: in (0,1) ;
var    a[1,1],...,a[r,n] shared array of boolean ;   {a[1..r,p] is owned by process p }

procedure α-TasOnce(p): boolean ;                                {p is the invoking process}
var    i, row: int ;
var    tmp[1..n]: array of boolean ;
begin
1:        for i ∈ {1..n} do
              tmp[i] := READ(a[1,i]) ;
              if tmp[i] = 1 then row := r; goto L2 fi          {game started/memory dirty}
          od ;
2:        row := 0 ;                                                      {join the game}
3 (L1):   row := row + 1 ; WRITE(a[row,p],1) ;                             {step forward}
4:        for i ∈ {1..n} − {p} do tmp[i] := READ(a[row,i]) ; od ; {check contention at row }
5:        if tmp(i) = 0 for all i ≠ p then                               {if alone at row}
6:            if row = r then return(Success) else goto L1 fi ;
          else
7:            if row = r then goto L2 ;
8:            else goto (L1,L2) with probability (s, 1 − s) fi ;
9 (L2):   while row > 0 do WRITE(a[row,p],0)) ; row := row − 1 ; od ;           {back-off}
10:       return(Failure) ;
end
```

Figure 1: Protocol α-TasOnce for q out of n processes (p is the invoking process)

instance processes b_1, b_2 and b_3 standing at row row and suppose that the adversary freezes the first two. It is possible for b_3 to step ahead and to be lucky at row $row + 1$ and that eventually b_3 and all other processes ahead of b_1 and b_2 back off. Doing this they all reinitialize their bits to zero (line 9 of the protocol in Figure 1). Afterwards, b_2 could be unfrozen by the adversary and be lucky at row $row + 1$ and back off later. And so on.

Claim 3 (Safety) *At most one process can win.*

Proof. A process that is lucky at row r will not back-off. In particular it will not clean up its row of bits (line 9 of the protocol). Hence, having two lucky processes at row r contradicts Claim 2. □

Claim 4 *Consider the set of executions such that no process crashes occur and such that the bits of the* α-Test&SetOnce *object are initialized correctly to 0. Then, the success probability* *of* α-TASONCE *with* $q \leq n$ *invoking processes is at least*

$$\alpha := \frac{2s}{1+s} - (n+q)\ s^r.$$

Proof. Intuitively, the aim of the adversary is to prevent a process from winning. We will bound the probability that the adversary succeeds by increasing its power. Since we assume that no crashes occur, there are only two ways for the adversary to prevent a win from occurring: Either two or more processes reach row r or all processes back off prior to row r. We make "two copies" of the game and allow the adversary to play both. That is, we consider two objects, each invoked by the same number of processes; in one game the adversary will try to maximize the probability that the first of the two "spoiling" events occurs and in the other it tries to maximize the probability of the second "spoiling" event. The adversary succeeds if it wins at least one of the two games. Clearly, this is an upper bound on the probability that it succeeds by playing just one game.

Consider the first case and focus on the subset C of processes that do not back-off. The adversary can bring one process p to row r with probability 1. What is the probability that *another* process $p' \in C$ reaches row r? (Processes not in C do not reach row r by definition.) By Claim 2, at each row at most one process in C can be lucky. Therefore p' reaches row r only if there is a sequence of coin tosses that brings some process p_1 from row 1 to row 2, another process p_2 from row 2 to row 3, and so on. These processes might be the same or different but, in any case, the probability of these consecutive successes is s^r. Hence the probability that the adversary spoils the game in this case is

$$\Pr[\text{some } p_i \neq p \text{ reaches row } r] \leq \sum_i \Pr[p_i \text{ reaches row } r] = q \Pr[p' \text{ reaches row } r] = qs^r.$$

Consider now the other case. Since we assume that no crashes occur, all participating processes must toss their coins until they either back off or reach row r. How long it takes is immaterial because the coin flips are independent. Since we are interested in the probability that they all back off before row r it is disadvantageous for the adversary to have some of the processes stepping forward with probability 1. Indeed, these probability 1 events only increase the number of forward steps of some processes. Hence, the probability of having no winner can be bounded as in the synchronous game, namely by $1 - f(2) + ns^r$. □

Setting $r := \log q$ the theorem is proven. □

The analysis above uses 1-writer n-reader 1-bit registers as intercommunication primitives. Of course, if we use more complex primitives then the complexity figures decrease.

Corollary 1 *For every* $0 < s < 1$*, there is an implementation of an* α-Test&SetOnce *object that succeeds with probability at least* $\alpha := 2s/(1+s) - o(1)$ *(provided no processes crash) and invoked by* q *out of* n *processes it uses* n *copies of* 1*-writer* n*-reader* $O(\log\log q)$*-bit shared read/write variables and its running time is* $O(n \log q)$ *read/writes of shared variables.*

Proof. We can replace each array $a[1..r, p]$ of α-TasOnce by a single $O(\log \log q)$-bit variable which is used as a counter which counts up to $r = c \log q$, and simplify the protocol in the obvious way. □

In [3] the notion of "composite register" or "snapshot object" is constructed from multi-user wait-free atomic read/write registers. A *composite register* is useful to obtain a "snapshot" of the states of a set (or all) shared variables in a system. It is a wait-free read/write register $R = (R_1, \ldots, R_m)$ where each R_i can be written by some process (without changing R_j $(j \neq i)$) and each process can atomically read all of $(R_1, \ldots, R_m)$. Since the atomic accesses are linearly ordered by definition each read by a process gives it a snapshot of the contents of all shared variables $R_1, \ldots, R_m$.

Corollary 2 *For every $0 < s < 1$, there is an implementation of an α-*Test&SetOnce *object, that succeeds with probability at least $\alpha := 2s/(1+s) - o(1)$ (provided no processes crash) and for q out of n processes it uses a single n-writer n-reader 1-writer-per-component n-component composite register with $\log \log n$-bit components and its running time is $O(\log q)$ read/writes on the composite register.*

Proof. The array of counters of the previous corollary can be replaced by a composite register, aka *snapshot object*, as defined in [3, 1]. This improves the complexity figures and would simplify the protocol, given the availability of a snapshot object implementation. □

4 A Wait-free Naming Protocol

We base our wait-free randomized naming protocol on the α-*Test&SetOnce* object. There are n competing processes $p_1, \ldots, p_n$ and the key space consists of m α-*Test&SetOnce* objects—one for each key.

4.1 Simple but Too Hard to Analyze Strategy

At first glance a simple strategy (as in Appendix B) may suffice: Each process repeatedly invokes an object selected uniformly at random, until it succeeds in getting a key (and no other process can get that key). On average, we expect αm objects to fire correctly in the sense that they assign their key to one of the invoking processes. By choosing $m := n/(\alpha\beta)$ to take care of random fluctuations, we can ensure that every process eventually gets a key.

The running time of this simple strategy seems hard to analyze. At any point in time, there will be a set of still competing processes to be matched with a set of available keys. The number of available objects determines the probability of getting a key (the randomly selected object must at least be available). In turn, this probability determines the number of rounds needed for the slowest process before it gets a key. The problem is that the number of empty objects at any given round depends on what the adversary does; processes can be

param n: **int** ; {number of processes}
param ϵ: **real** ; {specify key-range }
var m, l_s: **shared int** ; { key range $m := \lceil (1+3\epsilon)n \rceil$}
{ segment size $l_s := \lfloor c \cdot \log n \rfloor$}

procedure nameSEGMENT(): **int** $\in \{0..m\}$
var $start, key, l$: **int** $\in \{0..m\}$;
var $succeed$: **boolean** ;
begin
  $start := random \in \{1..m\}$;
  $l := \lfloor start/l_s \rfloor l_s + 1$; {beginning of segment }
  $key := start$; $succeed := 0$;
  repeat {Phase 1: try to get key within segment}
    $key := ((key+1) \bmod l_s) + l$;
    $succeed := (\alpha\text{-TASONCE}_{key}(p) = \text{SUCCESS})$; {Compete for key}
  until $succeed = 1$ **or** $key = start$;
  while $succeed = 0$ **do** {Phase 2: linear search}
    $key := (key+1) \bmod m$;
    $succeed := (\alpha\text{-TASONCE}_{key}(p) = \text{SUCCESS})$;
  od ;
  return(key) ;
end

Figure 2: Protocol SEGMENT for process p

stopped or let go to occupy an object. It is not clear to us how to frame all possible adversarial strategies.

4.2 Trickier but Easy to Analyze Strategy SEGMENT

By imposing just a little bit of structure on the way the objects can be invoked it is possible to come up with a simple and efficient protocol SEGMENT amenable to a clean analysis. Set

$$m := \frac{n}{\alpha\beta},$$

where α is the reliability of the α-*Test&SetOnce* object and β is a parameter which will take care of random fluctuations. We will show below that $\beta \approx (1-2\epsilon)$ for some other parameter ϵ to be determined later, where ϵ can be taken arbitrarily small (but must be fixed). Therefore, by setting $\alpha = 1-\epsilon$, we have $m \approx (1+3\epsilon)n$. We divide the key space into *segments*, each of length

$$l_s = c \ln n$$

where c is a constant to be specified later and "ln" denotes the natural logarithm. We think of each segment as a ring of objects, where the i-th and the $(i+l_s)$-th objects in a segment are the same. The protocol is shown in Figure 2 and is as follows. Each process selects a random key $start \in \{1, \ldots, m\}$; this automatically determines a segment whose initial position we denote by l. The processes will then start invoking keys by "walking" around the segment, that is, a process will first try to get a key by invoking the *α-Test&SetOnce* object corresponding to its first random choice *start*; then, if necessary, it will invoke the next (modulo l_s) object in the ring, and so on, until it gets back to the starting point *start*. As we shall see, with high probability, every process will get a key before reaching this point. In the extremely unlikely event that some process will not find a key in its segment, the whole key range is scanned repeatedly until a key is found (Phase 2 of the protocol). This will ensure that all processes eventually get a name.

Lemma 3 *For every* $0 < \alpha, \beta < 1$, *protocol* SEGMENT *solves the naming problem for* n *processes using* $m = n/(\alpha\beta)$ α-Test&SetOnce *objects. The protocol is safe and correct. With probability* $1 - o(1)$, *the running time is* $O(n \log n \log\log n)$ *read/writes to 1-writer,* n*-reader shared atomic bits.*

Proof. We show that, with high probability, all processes in a segment will be *captured*— they will find their key or crash inside the segment. Therefore, every *α-Test&SetOnce* object is invoked $O(l_s) = O(\log n)$ times with high probability as well. Consequently, we can apply Theorem 1 with $q := O(\log n)$. For non-faulty processes this means that they will find the key within the segment. First, we show that the processes distribute evenly among the segments. Let

$$P_s = (\#\text{ processes in segment } s).$$

Then,

$$\Pr[\text{process } p \text{ selects } s] = \frac{l_s}{m} = c\alpha\beta\frac{\ln n}{n}$$

and

$$\mu_s := \mathbf{E}[P_s] = \sum_p \Pr[\text{process } p \text{ selects } s] = c\alpha\beta \ln n$$

Since the segments are chosen independently we can invoke the Chernoff-bounds to estimate the tails of the Binomial distribution in the following form (see for example [22]):

$$\Pr[|P_s - \mu_s| > \epsilon\mu_s] \leq 2e^{-\epsilon^2\mu_s/3} = 2n^{-\epsilon^2 c\alpha\beta/3}$$

By setting

$$c \geq \frac{6}{\alpha\beta\epsilon^2} \tag{4}$$

we can ensure that

$$\Pr[P_s > (1+\epsilon)\mu_s, \text{ for some } s] \leq \sum_s \Pr[P_s > (1+\epsilon)\mu_s] \leq 2nn^{-\epsilon^2 c\alpha\beta/3} < \frac{2}{n} \tag{5}$$

so that the probability that *some* segment receives more that $(1+\epsilon)\mu_s$ processes is $2/n = o(1)$.

We also need to ensure that every segment captures all of its processes. Here we need to take care of the adversary. Basically the problem is as follows. Whenever an object is invoked, the adversary may or may not crash a process during its object invocation; when this happens we say that the object is *corrupt*. Consider the case when one process walks around the whole segment without finding a key. When this happens all objects in the segment are invoked. If a is the number of corrupt objects then, each of the $(l_s - a)$-many non-corrupt objects succeeds with probability α independently of other objects. In other words, we are considering $l_s - a$ Bernoulli trials with probability of success equal to α, where "success" means that some of the invoking processes is given a key. Notice that for small values of $l_s - a$, large deviations from the mean are more likely. Therefore, it is advantageous for the adversary to crash processes, thereby corrupting objects, in the hope that some of the segments will not capture all of its processes (while our aim is to ensure that all segments will capture their processes). We now show that with an appropriate choice of the constant c this almost surely never happens.

With the above notation, and recalling our definition of captured, the expected number of captured processes is at least

$$a + \alpha(l_s - a).$$

"At least" because for each corrupt object the adversary must crash at least one process. By the Chernoff bounds, the true number of captured processes is at least

$$a + (1-\epsilon)\alpha(l_s - a).$$

with probability at least

$$1 - 2\exp\left\{-\epsilon^2\alpha(l_s - a)/3\right\}.$$

We know that with high probability each segment has at most $P_s = (1+\epsilon)c\alpha\beta\ln n$ processes. A straightforward computation shows that $P_s \leq a + (1-\epsilon)\alpha(l_s - a)$ for every $a \geq 0$ as long as

$$\beta = \frac{1-\epsilon}{1+\epsilon} \approx 1 - 2\epsilon.$$

What is left to verify is that, no matter how a is chosen by the adversary, all segments capture their processes with high probability. To this end, notice that $a \leq P_s$ and therefore $l_s - a \geq l_s - P_s$, which implies that the probability that a segment fails to capture its processes is at most

$$2\exp\left\{-\epsilon^2\alpha(l_s - P_s)/3\right\}$$

a bound which is independent of the adversary. A straightforward computation shows that this exceptional probability is at most $2/n^2$ provided that

$$c \geq \frac{6}{(2-\epsilon)\alpha\epsilon^3}$$

Since there are $m/l_s < n$ segments, the probability that some segment fails is $2/n = o(1)$. Together with Equation 5, this gives that with probability $1 - o(1)$ each process finds a key within

$O(\log n)$ object invocations. (Similarly, every object is invoked $O(l_s) = O(\log n)$ times with probability $1 - o(1)$.) By Theorem 1 every object invocation has running time $O(n \log\log n)$ reads/writes to 1-writer, n-reader 1-bit atomic registers. Thus, the *maximum running time* among all non-crashing processes is $O(n \log n \log\log n)$ bit operations with probability $1-o(1)$.

But is the protocol safe: does every process obtain a distinct key under every circumstance? If a process fails to find a key in its segment it scans the whole key space until a key is found. We saw in Section 3 that the α-*Test&SetOnce* objects are safe, they never give the key to more than one process. Since there are more objects than processes and non-corrupt objects can be invoked repeatedly until they assign the key, sooner or later every correct process will find a key (with probability 1). The lemma is proven. □

Lemma 3 does not imply that the average running time over all coin flip sequences of outcomes used by the processes involved (the expected running time) is $O(n \log n \log\log n)$ bit operations—the expected running time may still be infinite. This expectation has to be bounded to meet our definition of "wait-freeness" in Appendix A.

To achieve a bounded *expected* running time we need to use $O(n \log n)$ bit operations per object invocation, rather than $O(n \log\log n)$. To see the problem, recall that Theorem 1 states that the object succeeds with probability α, provided $O(n \log q)$ bits are used, where q is the number of competing processes. If $q = \Theta(n)$ then $O(n \log n)$ bits must be used (or otherwise the bound given by Lemma 2 becomes worthless). Although a very unlikely event, it is entirely possible that linearly many processes fail in their segment and start scanning the whole key space. In such cases, the average running time will be high because it would take an exponentially long time before each of the scanning processes gets a key. But if we are willing to use $O(n \log n)$ bits per α-*Test&SetOnce* object, the average running time will still be only $O(n \log^2 n)$ bit operations.

Theorem 2 *For every $0 < \alpha, \beta < 1$, protocol* SEGMENT *solves the naming problem for n processes using $m = n/\alpha\beta$ α-*Test&SetOnce *objects. The protocol is wait-free, safe and correct. The expected running time is $O(n \log^2 n)$ read/writes to 1-writer, n-reader shared atomic bits.*

Proof. As we saw in the proof of Lemma 3, the probability that a process has to resort to scanning the whole key space is $o(1)$. If we denote by a the total number of corrupt keys, then by the time the process has scanned the whole space there have been $m - a > 3\epsilon n$ non corrupt objects, each firing independently with probability α. Then, with probability at least

$$1 - 2\exp\{-\delta^2 \alpha \epsilon n\}$$

(a bound independent of a) at least $(1 - \delta)\alpha(m - a) \geq n - a$ object are assigned a key, implying that each of the $m - a$ correct processes receives a (unique) key. Define $p^{kn} := 2\ \exp\{-\delta^2 \alpha \epsilon k n/3\}$. Then, with probability at most p^{2n} a second scan is needed, and so on.

The average running time, in bit operations, is at most

$$O(n \log^2 n)(1 - o(1)) + o(1)n \sum_{k>0} \frac{k}{p^{kn}} = O(n \log^2 n).$$

It is clear that the above together with Claim 3 and Claim 1 imply that protocol SEGMENT is a wait-free solution for the process naming problem even in the average sense. The theorem is proven. □

Remark 1 In practice the protocol will be much faster for most of the keys, because the expected number of processes per object after the first random object is selected is $n/m < 1$. Also, a very large fraction of the processes will need just one invocation to get a key; well-known results on martingale inequalities state that when n processes select a random key out of m keys, the fraction of keys chosen by some process is very nearly $n(1-e^{-m/n}) > n(1-1/e)$. Hence, with high probability, very nearly $\alpha n(1 - 1/e)$ processes will get a key after just one invocation of an α-*Test&SetOnce* object.

Remark 2 Similar results hold if we implement the α-*Test&SetOnce* object with 1-writer n-reader $O(\log \log n)$-bit shared read/write variables or n-writer n-reader 1-writer-per-component n-component composite registers with $\log \log n$-bit components (as in Corollaries 1, 2).

Acknowledgment

We thank the referees for their constructive comments which resulted in a substantial improvement of the presentation.

A System Execution, Adversary, Computational Complexity

A system *execution* is an infinite sequence $\mathcal{E} := c_0 s_1 c_1 \ldots$ of alternating steps s_i and states c_i satisfying that each s_i is enabled in state c_{i-1} and c_i is the configuration of the system after the execution of s_i, for all $i > 0$. Technically, when a process halts it enters infinitely many times a distinguished *idle* state c_∞ through an idle step s_∞. All registers are initialized to zero contents in the unique start state c_0. If we initialize with "dirty shared memory" then all registers can have arbitrary initial contents. The set of all system executions is denoted by Ω.

An adversary is best explained by identifying it with a conditional probability density function $\mathcal{A}(s_i c_i | \mathcal{E}_{i-1})$ where $\mathcal{E}_{i-1} := c_0 s_1 \ldots c_{i-1}$ is an initial segment of $\mathcal{E}$, step s_i is enabled in state c_{i-1}, and c_i is the state resulting from executing step s_i in state c_{i-1}, for $i > 0$. Now $\mathcal{A}(s_i c_i | \mathcal{E}_{i-1})$ is the probability that the initial execution segment $\mathcal{E}_i = \mathcal{E}_{i-1} s_i c_i$ is realized given that $\mathcal{E}_{i-1}$ has happened. If the adversary is randomized itself then we have $\sum_s \sum_{c_s} \mathcal{A}(s c_s | \mathcal{E}_{i-1}) = 1$ with the summation taken over the different enabled steps s in state

c_{i-1} and the states c_s that can result from step s: a single state if s is not randomized and more states if s is a randomized step (a coin flip). If the adversary is deterministic then it chooses deterministically a step s and $\sum_{c_s} \mathcal{A}(sc_s|\mathcal{E}_{i-1}) = 1$.

Starting from $\mathcal{E}_0 := c_0$ the adversary induces a measure $\mathcal{A}$ over all legal system executions $\mathcal{E}$ defined by $\mathcal{A}(\mathcal{E}) := \lim_{i\to\infty} \mathcal{A}(\mathcal{E}_i)$ [2] where $\mathcal{A}(\mathcal{E}_i) := \mathcal{A}(s_i c_i|\mathcal{E}_{i-1})\mathcal{A}(\mathcal{E}_{i-1})$ and $\mathcal{E}_i = \mathcal{E}_{i-1}s_i c_i$, for $i > 0$. The adversary is "adaptive" since it schedules the process executing the next step based on the complete knowledge of the initial segment of the system execution including the random outcomes of past coin flips. It can arbitrarily delay processes or even *crash* them by not executing enabled steps of particular processes. Below we express the strongest adversary (adaptive, with infinite computing power, and so on) as a probability measure on the set of executions as in [28]. Without loss of generality we assume that the only randomized steps the protocols use are fair coin flips.

Definition 2 Assume the above notation. An *adaptive adversary* is a probability measure $\mathcal{A}$ on Ω satisfying:

1. $\mathcal{A}(\mathcal{E}_0) = 1$, where $\mathcal{E}_0$ is the initial execution segment;
2. $\mathcal{A}(\mathcal{E}_i) = \sum_{s,c} \mathcal{A}(\mathcal{E}_i sc)$, where the summation is over enabled steps s in state c_i and the state(s) c resulting from executing step s in state c_i;
3. $\mathcal{A}(\mathcal{E}_i sc_h) = \mathcal{A}(\mathcal{E}_i sc_t)$, for each coin-flip step s with c_h is the state resulting from c_i when the outcome of s is "heads" and c_t is the state resulting from c_i when the outcome of s is "tails.".

The first two conditions—already implied by the notion of probability measure—are included for completeness. The third condition ensures that the adversary has no control over the outcome of a fair coin flip: both outcomes are equally likely. This definition is readily generalized to biased coins and multi-branch decisions. Now that adversaries have been defined, we can define the expected length $\mathbf{E}(\mathcal{E}_i, j)$ of process p_j's final execution following a finite initial execution segment $\mathcal{E}_i$. Let $\mathcal{E}$ be an infinite execution starting with $\mathcal{E}_i$. Let $l_{\mathcal{E}_i,j}(\mathcal{E})$ be the number of non-idle steps of process p_j following $\mathcal{E}_i$ in $\mathcal{E}$.

Definition 3 Assume the above notation. Define

$$\mathbf{E}(\mathcal{E}_i, j) = \sum_{k=1}^{\infty} k \cdot \frac{\mathcal{A}(\{\mathcal{E} \in S : l_{\mathcal{E}_i,j}(\mathcal{E}) = k\})}{\mathcal{A}(\mathcal{E}_i)}.$$

Since the summation includes the case $k = \infty$ the expected length is infinite if (but not necessarily only if) the set of infinite histories in which an operation execution has infinitely many

[2] With $\mathcal{E}_i$ denoting a finite initial segment of an execution and Ω the set of all infinite executions $\mathcal{E}$, the traditional notation is "$\mathcal{A}(\Gamma_{\mathcal{E}_i})$" instead of "$\mathcal{A}(\mathcal{E}_i)$" where *cylinder* $\Gamma_{\mathcal{E}_i} = \{\mathcal{E} \in S : \mathcal{E}$ starts with $\mathcal{E}_i\}$. We use "$\mathcal{A}(\mathcal{E}_i)$" for convenience.

events, has positive measure. The normalization w.r.t. $\mathcal{E}_i$ gives the adversary a free choice of 'starting' configuration. The *running time* of a deterministic protocol is the maximum number of non-idle steps, taken over all legal executions, executed by a non faulty process.

Definition 4 An implementation of a concurrent object shared between n processes is *wait-free*, if there is a finite bound $f(n)$ such that for all adversaries $\mathcal{A}$ and for all $\mathcal{E}_i, j$, the expected length $\mathbf{E}(\mathcal{E}_i, j) \leq f(n)$.

B Simple Approach Does Not Work

A related observation was made with respect to the symmetric communication model in [10]. In our case we use the Method of Bounded Differences (MOBD) [21]. Suppose we have n independent random variables X_i each taking values in a finite set A_i and let $Y = f(X_1, \ldots, X_n)$ be a measurable function. If, for all vectors A and B differing only in the i-th coordinate,

$$|f(A) - f(B)| \leq c_i$$

then

$$\Pr\left[|Y - \mu| > \epsilon\mu\right] \leq 2e^{-2\epsilon^2\mu^2/\sum_i c_i^2} \tag{6}$$

where $\mu = \mathbf{E}[Y]$. We will use this in a ball-and-bin scenario, where X_i denotes the bin where ball i ends up and Y will measure things such as the number of bins with exactly k balls, the number of bins with at least k balls, and the like. In these cases, it easy to see that $c_i = 1$ for all i and the bound becomes

$$\Pr\left[|Y - \mu| > \epsilon\mu\right] \leq 2e^{-2\epsilon^2\mu^2/n}.$$

We have n processes and a name space of size $m = (1 + c)n$ $(0 \leq c \leq 1)$. For a naming algorithm to be good, we want both c and the running time to be as small as possible.

The most obvious naming algorithm works as follows: Every process chooses uniformly and independently a *tentative* random key and checks whether it is the only process claiming that key. If so, the process secures the key. Otherwise, it tries another random key, and so on.

To check whether a process is the only claimant for a key we use the following mechanism. For each key k there is an array $b[k, 1..n]$ where bit $b[k, p]$ is owned by process p $(1 \leq p \leq n)$. All bits can be read by all processes. Upon choosing a specific key value k, a process sets its own bit $b[k, p]$ to 1 and subsequently reads the other bits of the array $b[k, 1..n]$ to see whether it is the only claimant, Figure 3. If a process was alone a SUCCESS is returned, otherwise a FAILURE. Notice that the bit $b[k, p]$ is reset to 0 in case of failure so that a process can try again.

It is easy to verify that this solution is safe in the sense that no two processes ever get the same key. It is more difficult to see that its running time is unsatisfactory: for $c < 1$ there

```
param n: int ;                                                        {number of processes}
param c: real ∈ (0,1) ;                                               {specifies key range}
var   m: shared int ;                                             { key range m := ⌊n(1+c)⌋}
var   b[1..m,1],...,b[1..m,n] shared array of boolean ;
                                                           {each b[1..m,p] is owned by process p}
procedure alone(key): boolean ;                                       {key = candidate name}
begin
   WRITE(b[key,p],1) ;
   for i ∈ {1..n} do
        if READ(b[key,i])=1 then WRITE(b[key,p],0); return(FAILURE) fi {not alone}
   od ;
   return(SUCCESS) ;                                                             {alone}
end
procedure simp-name(): int ∈ {0..m}
begin
   repeat key := random ∈ {1..m} until alone(key) = 1; return(key) ;
end
```

Figure 3: A simple approach to naming: protocol for process p

are adversarial strategies that force some process to take exponentially many steps with high probability. The problem is that the adversary knows the key k chosen by a process p before p executes its subsequent $WRITE(b[k,p],1)$ step. Therefore, the adversary can postpone the execution of this step until some other process q chooses the same key k. At this point, the adversary schedules the steps of p and q such that both of them don't secure key k.

Adversarial strategy: If step $WRITE(b[key,p],1)$ is enabled for process p but the adversary delays execution then we say that p is *frozen*. If a p has chosen k but has not yet executed $WRITE(b[k,p],1)$ we say that the process is *claiming* k. Let $\lambda = 1/(1+c)$ so that $n = \lambda m$, and fix some $\epsilon < \lambda$.

The adversary schedules all processes in turn to perform their first random choices. Define event A as "at least ϵn keys are chosen by exactly two processes." A standard application of the MOBD above shows that the probability that A does not occur is at most $e^{-c_1 n}$, where c_1 is a constant depending only on ϵ and λ. The adversary selects ϵn such keys and freezes the set F_1 of corresponding processes. The adversary schedules the operations of the remaining processes until each of them claims one of the remaining keys and no such key is claimed by more than one such process. (If more than one process claims one of these keys the adversary schedules events such that all but one of them back off and try again until they are unique claimants for other keys.) At this point, there are at least $(1-2\epsilon)n = (1-2\epsilon)\lambda m$ keys that are claimed by a unique process. Call these the *red* keys. Now the processes in F_1 are unfrozen. The adversary schedules their operations so that their first attempts fail and all of them do a second tentative random key choice. Define event B as "ϵn red keys are claimed by exactly one process in F_1" and let F_2 be the set of processes claiming those keys (each such key is now claimed by exactly two processes). Then $|F_2| = |F_1|$ and the adversary can repeat the scenario with F_2 substituted for F_1. A tedious, but standard, application of the MOBD shows that the probability that B does not occur is at most $e^{-c_2 n}$, where, again, c_2 is a constant depending only on ϵ and λ. Therefore, with high probability the adversary will be able to force some process to try an exponential number of keys.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, Atomic snapshots of shared memory. *J. Assoc. Comp. Mach.*, 40:4(1993), 873–890.

[2] Y. Afek, E. Gafni, J. Tromp, and P.M.B. Vitányi, Wait-free test-and-set In *Proceedings of the 6th International Workshop on Distributed Algorithms*, vol. 647, *Lecture Notes in Computer Science*, Springer-Verlag 1992, pp. 85–94.

[3] J. Anderson, Composite registers. *Distributed Computing* 6(1993), 141-154.

[4] J. Aspnes and M. Herlihy, Fast Randomized Consensus Using Shared Memory. *Journal of Algorithms* 11(1990), 441-461.

[5] J. Aspnes and O. Waarts, Randomized Consensus in Expected $O(n \log^2 n)$ Operations Per Processor. In *Proceedings of FOCS 1992*, pp. 137-146.

[6] H. Attiya, A. Bar-noy, D. Dolev, D. Peleg, and R. Reischuk, Renaming in an Asynchronous Environment. *J. Assoc. Comput. Mach.*, 37:3(1990), 524–548.

[7] A. Bar-Noy and D. Dolev, Shared Memory vs. Message-passing in an Asynchronous Distributed Environment. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*, 1989, pp. 307–318.

[8] E. Borowsky and E. Gafni, Immediate Atomic Snapshots and Fast Renaming. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, 1993, pp. 41–52.

[9] H. Buhrman, J.A. Garay, J.H. Hoepman, and M. Moir, Long-Lived Renaming Made Fast, In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, 1995, pp. 194–203.

[10] O. Eğecioğlu and A.K. Singh, Naming Symmetric Processes Using Shared Variables. *Distributed Computing*, 8:1(1994), 1–18.

[11] M.J. Fischer, N.A. Lynch, and M.S. Paterson, Impossibility of Distributed Consensus with One Faulty Processor. *J. Assoc. Comput. Mach.* 32:2(1985), 374–382.

[12] N.A. Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1996.

[13] M. Herlihy, Wait-free synchronization. *ACM Trans. Progr. Lang. Syst.*, 13:1(1991), 124–149.

[14] M. Herlihy, Randomized Wait-Free Concurrent Objects. In *Proc. 10th ACM Symp. Principles Distrib. Comput.*, 1991, pp. 11–21.

[15] M. Herlihy and N. Shavit, The Asynchronous Computability Theorem for t-Resilient Tasks. In *Proc. 25th ACM Symp. Theory of Computing*, 1993, pp. 111–120.

[16] S. Kutten, R. Ostrovsky, and B. Patt-Shamir, The Las-Vegas Processor Identity Problem (How and When to Be Unique). In *Proc. 2nd Israel Symp. Theor. Comput. Syst.*, IEEE Computer Society Press, 1993.

[17] L. Lamport, On Interprocess Communication. *Distributed Computing*, 1:1(1986), 86–101.

[18] M. Li, J. Tromp, and P.M.B. Vitányi, How to Share Concurrent Wait-free Variables, *J. Assoc. Comput. Mach.*, 43:4(1996), 723–746.

[19] R.J. Lipton and A. Park, Solving the processor identity problem in $O(n)$ space, *Inform. Process. Lett.*, 36(1990), 91–94.

[20] M.C. Loui and H.H. Abu-Amara, Memory Requirements for Agreement Among Unreliable Asynchronous Processes. *Advances in Computer Research*, vol. 4, JAI Press, Inc. 1987, pp. 163–183.

[21] C. McDiarmid, On the method of bounded differences. *Surveys in Combinatorics*, J.Siemons ed., London Math. Society Lecture Note Series **141** (1989), 148–188.

[22] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.

[23] M.O. Rabin, The Choice Coordination Problem. *Acta Informatica 17*, (1982), 121–134.

[24] M. Saks, N. Shavit, and H. Woll, Optimal Time Randomized Consensus- Making Resilient Algorithms Fast in Practice. In *Proc. SIAM-ACM Symp. Data-Struct. and Algor.*, 1991, pp. 351-362.

[25] A.K. Singh, J.H. Anderson, and M.G. Gouda, The Elusive Atomic Register Revisited, *J. Assoc. Comput. Mach.*, 41:2(1994), 311–339.

[26] A. Tanenbaum, *Computer Networks*. Prentice-Hall, 1981.

[27] J. Tromp, How to Construct an Atomic Variable. In *Proc. 3rd Int'l Workshop Distribut. Algor., Lecture Notes in Computer Science, Vol. 392*, Springer-Verlag, Heidelberg, 1989, pp. 492–302.

[28] J. Tromp and P. Vitányi, Randomized Wait-Free Test-and-Set, CWI Tech. Report CS-R9113, Amsterdam, March 1991, Submitted.