On the Importance of Having an Identity or, is Consensus really Universal? (Extended Abstract)

Harry Buhrman[1], Alessandro Panconesi[2], Riccardo Silvestri[3], and Paul Vitanyi1

[1] CWI, Amsterdam
[2] Informatica, Universitá di Bologna,
[3] Informatica, Universitá de L'Aquila

Abstract. We show that Naming– the existence of distinct IDs known to all– is a necessary assumption of Herlihy's universality result for Consensus. We then show in a very precise sense that Naming is harder than Consensus and bring to the surface some important differences existing between popular shared memory models which usually remain unnoticed.

1 Introduction

The consensus problem enjoys a well-deserved reputation in the (theoretical) distributed computing community. Among others, a seminal paper of Herlihy added further evidence in support of the claim that consensus is indeed a key theoretical construct [12]. Herlihy's paper considers the following problem: Suppose that, besides a shared memory, the hardware of our asynchronous, parallel machine is equipped with objects (instantiations) of certain abstract data types $T_1, T_2, \ldots, T_k$; given this, is it possible to implement objects of a new abstract data type Y in a wait-free manner? This question is the starting point of an interesting theory leading to many results and further intriguing questions (see [12, 14] among others). Roughly stated, one of the basic results of this theory, already contained in the original article of Herlihy, is this: If an abstract data type X, together with a shared memory, is powerful enough to implement consensus for n processes in a wait-free manner then, X, together with a shared memory, is also powerful enough to implement in a wait-free manner for n processes any other data structure Y. This is Herlihy's celebrated universality result for consensus.

In this paper we perform an analysis of some of the basic assumptions underlying Herlihy's result and discover several interesting facts which, in view of the above, are somewhat counter-intuitive and that could be provocatively be summarized by the slogans "consensus without naming is not universal" and "naming with randomization is universal." To state our results precisely we shall recall some definitions and known results.

The **naming** problem is as follows: Devise a protocol for a set of n processes such that, at the end, each non faulty process has selected a unique identifier (key). If processes have identifiers to start with then we have the **renaming** problem. Besides time and space, the size of the *name space* too– the set of possible identifiers– is considered to be a resource whose consumption is to be minimized.

We shall concern ourselves with **probabilistic protocols**– every process in the system, modelled as an i/o automaton, has access to its own source of unbiased random bits– for systems consisting of **asynchronous** processes communicating via a **shared memory**. Processes can suffer from **crash failures**. The availability of objects of abstract data type `consensus` and `naming` is assumed.

The protocols we devise are **wait-free** (i.e. $(n-1)$-resilient) in spite of the **adversary**, the "malicious" non-deterministic agent (algorithm) modeling the environment. The adversary decides which, among the currently pending operations, goes on next. Pessimistically one assumes that the adversary is actually trying to force the protocol to work incorrectly and that the next scheduling decision– which process moves next– can be based on the whole past history of the protocol execution so far. This is the so-called *adaptive* or **strong** adversary. In contrast, sometimes it is assumed that the adversary decides the entire execution schedule beforehand. This is the so-called *oblivious* or **weak** adversary.

In the literature two shared-memory models are widespread. The first assumes `multiple reader - multiple writer` registers, in which every location of the shared memory can be written or read by any process. The other model assumes `multiple reader - single writer` registers. Here, every register is owned by some unique process, which is the only process that can write on that register, while every process is allowed to read the contents of any register. 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, 13]), 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 non-trivial, we assume that each process p accesses the n register by means of a permutation π_p. That is, register π_p^i– p's ith register– will always be the same register, but, for $p \neq q$, π_p^i and π_q^i might very well differ. Besides making the problem nontrivial, this models certain situations in large dynamically changing systems where the consistency requirement is difficult or impossible to maintain [17] or in cryptographical systems where this kind of consistency is to be avoided. In both models reads and writes are atomic operations; in case of concurrent access to the same register it is assumed that the adversary "complies with" some non-deterministic, but fair, policy. In this paper we shall refer to the first as the **symmetric** memory model and to the second as the **asymmetric** memory model. It is worth pointing out that the first model is considered to be more "powerful" than the second. As we shall see, this intuition is somewhat misleading.

In this paper we show the following. *Assume that each processes has access to its own private source of independent random bits.* Then,

– **Consensus is "easy":** Assuming that (a) the memory is symmetric, and (b) processors are identical i/o automata without identifiers then, there exist wait-free, Las Vegas consensus protocols for n process, for any $n > 1$, which work correctly even assuming the strong adversary. In fact, we exhibit a protocol whose running time (per processor) is polynomial both in expectation and with high probability.

– **Naming is "hard":** In contrast, if (a) the memory is symmetric, (b) processors are identical i/o automata without identifiers which have access to (c) consensus objects then, Las Vegas naming is impossible, even assuming the weak adversary. Note that Montecarlo naming is trivial– it is enough that each process generate $O(\log n)$ many random bits and with probability $1 - o(1)$ no two of them will be identical. [1]
– **Naming + Symmetry = Asymmetry:** If the memory is symmetric and processes have unique identifiers then the memory can be trivially made asymmetric. In this paper we show the other direction of the equivalence above namely, if the memory is asymmetric then, naming *is* possible even against the strong adversary. We exhibit a simple, modular protocol whose running time and space per process are polynomial in n, the number of processors, both in expectation and with high probability. The size of the name space is optimal, thereby improving upon a previous result of [19].

These results show that in a very precise sense, somewhat surprisingly, naming is harder, or perhaps more "fundamental", than consensus. As the second result shows, naming is impossible in a model richer than a model in which consensus is possible. Therefore naming, or some other form of asymmetry, is a necessary assumption in Herlihy's universality result. An inspection of Herlihy's construction does show that the assumption that processes have unique identifiers known to all, i.e. naming, plays a crucial role [12]. One might wonder whether in his model (deterministic, symmetric memory) names can be created from scratch. As we show in this paper, the answer is *"no!"*. In fat, they cannot even be generated if randomness (and consensus objects) are allowed. In view of this it would be more precise to restate Herlihy's result as "consensus plus naming is universal". Notice that, as the third result in the above list shows, if randomness is allowed then, asymmetric memory is powerful enough for naming. Thus, we have yet another indication that randomness increases the power of distributed systems as far as fault-tolerance is concerned.

A fundamental problem to confront with when dealing with parallel or distributed computation is "symmetry breaking." It is well-known that randomness is quite helpful in breaking the symmetry and that identifiers are another effective means to deal with the problem. Our results show that randomness is enough for consensus, a supposedly universal construct, but not for naming. Furthermore, they show that single-writer registers are inherently symmetry-breaking, whereas consensus is not. As a byproduct of our analysis we show that Herlihy's universality result does not apply to another, quite basic data type, unless naming or some other form of asymmetry is assumed. The data type in

[1] Recall that a Las Vegas protocol is always correct and that only the running time is a random variable, while for a Montecarlo protocol correctness too is a random variable.

question, which we call `selectWinner`, selects a unique winner among a set of n invoking processes. This task is impossible with symmetric memory, even if randomness and consensus are available. This result also shows in a different way how the power of randomization to "break the symmetry" is limited.

Interestingly, all randomized consensus algorithms existing in the literature known (to the authors) assume either asymmetric memory or the existence of identifiers (see, for instance, [1, 3, 4, 11]). Our results indicate that these assumptions are not necessary and that randomness is enough to create the "necessary asymmetry" as far as consensus is concerned.

Recently Chandra developed a very fast algorithm for consensus (whose asymptotic performance was subsequently improved by Aumann [9, 6]). His algorithm is much faster than the lower-bound shown by Aspnes for consensus [2]. How is this possible? Aspnes result holds for the asymmetric model, while Chandra uses several assumptions which, a priori, could be responsible for the speed-up: the availability of `multiple reader - multiple writer` registers instead of `multiple reader - single writer` registers– i.e. symmetric instead of asymmetric memory; pre-existing identifiers, and the **intermediate** adversary. This is a third kind of adversary, lying between the weak and the strong. Its behaviour is adaptive, but it has limited access to the outcome of the coin flips in that it can read the outcome of a coin flip only when this is read by some process, and not when the bit is generated (see, among others, [9]).

At first it would seem that the naming assumption must be the least important, perhaps even superfluous, for `multiple reader - multiple writer` registers certainly can cause a speed-up, and assuming a weaker adversary can circumvent impossibility results such as that of Aspnes. In fact, our results shows that, as far as Chandra's protocol is concerned, naming is necessary, for identifiers cannot be generated from scratch in his model. Aumann showed that the lower bound of Aspnes can be circumvented even assuming asymmetrc memory, i.e. `single writer- multiple reader` registers [6] We leave it as an open question whether the same holds without the naming assumption.

Our results also show that the widespread intuition that `multiple reader - multiple writer` registers are more "powerful" than `multiple reader - single writer` registers is somewhat misleading. The intuition is correct under the (quite reasonable) assumption that processes. It is however worth pointing out that the intuition is wrong without this assumption, for naming can be solved with `multiple reader - single writer` registers, but it is impossible with `multiple reader - multiple writer` registers, even if randomness is allowed. This highlights an important difference between these two models which, we feel, it is often overlooked and show that memory assumptions must be carefully stated.

Our first result, a randomized, wait-free protocol for consensus in the symmetric model (which can withstand the strong adversary) is obtained by combining several known ideas and protocols, in particular those in [3] and [9]. When compared to the protocol in [3] it is, we believe, simpler, and its correctness is easier to establish (see [20]). Moreover, it works in the less powerful symmetric model and can deal with the strong adversary, whereas the protocol in [9] works only against the intermediate adversary. From the technical point of view, our second result is essentially contained in [16] to which we refer for other interesting related results. Related work can be found in [5, ?].

In spite of the fact that we make use of several known technical ingredients, our analysis, we believe, is novel and brings to light for the first time new and, we hope, interesting aspects of fundamental concepts.

2 Consensus is easy, Naming is hard

We start by outlining a consensus protocol assuming that (a) the memory is symmetric, (b) processes are i/o automata **without** identifiers which have access to their own source of (c) random bits. Our protocol is obtained by combining together several known ideas and by adapting them to our setting. The protocol, a randomized implementation of n-process binary consensus for symmetric memory, is a modification of the protocol proposed by Chandra [9]. The original protocol cannot be used in our setting since its shared coins require that processes have unique IDs. Thus, we combine it with a modification of the weak shared coin protocol of Aspnes and Herlihy [3]. The latter cannot be directly used in our setting either, since it requires asymmetric memory. Another difference is that, unlike in Chandra's protocol, we cannot revert to Aspnes' consensus [1]. In this paper we are only interested in establishing the existence of a polynomial protocol and make no attempt at optimization. Since the expected running time of our protocol is polynomial, by Markov's Inequality, it follows that the running time and, consequently, the space used are polynomial with high probability (inverse polynomial probability of failure). Conceivably superpolynomial space could be needed. We leave it as an open problem whether this is necessary. In the sequel we will assume familiarity with the notion of weak shared coin of [3] to which the reader is referred.

The protocol, shown in Figure 1, is based on the following idea. Processes engage in a race of sorts by splitting into two groups: those supporting the 0 value and those supporting the 1 value. At the beginning membership in the two "teams" is decided by the input bits. Corresponding to each team there is a "counter", implemented with a row of contiguous "flags"– the array of booleans MARK[,]– which are to be raised one after the other starting from the left by the team members, cooperatively and asynchronously. The variable $position_p$ of each process records the rightmost (raised) flag of its team the process knows about. The protocol keeps executing the following loop, until a decision is made. The current team of a process is defined by the variable $estimate_p$. The process first increments its own team counter by raising the $position_p$-th flag of its own team (this might have already been done by some other team member, but never mind). This means that, as far as the process is concerned, the value of its own team counter is $position_p$ (of course, this might not accurately reflect the real situation). The process then "reads" the other counter by looking at the other team's row of flags at positions $position_p + 1, position_p, position_p - 1,$

in this order. There are four cases to consider: (a) if the other team is ahead the process sets the variable $newEstimate_p$ to the other team; (b) if the two counters are equal, the process flips a fair coin $X \in \{0,1\}$ by invoking the protocol $\text{GETCOIN}_\delta(,)$ and sets $newEstimate_p$ to X; (c) if the other team trails by one, the process sticks to its team, and (d) if the other team trails by two (or more) the process decides on its own team and stops executing the protocol. Before executing the next iteration, the process checks again the counter of its own team. If this has been changed in the meanwhile (i.e. if the $(position_p+1)$-st flag has been raised) then the process sticks to his old team and continues; otherwise, it does join the team specified by $newEstimate_p$ (which in case of a random coin flip can still be the old team). The array $\text{MARK}[i, s]$ implemented with `multiple reader - multiple writer` registers, while the other variables are local to each process and accessible to it only.

A crucial difference between our protocol and that of Chandra concerns procedure $\text{GETCOIN}_\delta(,)$. In Chandra's setting essentially it is possible to implement "via software" a *global coin*, thanks to the naming assumption. In the implementation in Figure 1, we use a protocol for a weak shared coin for symmetric memory. For every $b \in \{0,1\}$ and every $i \geq 1$ an independent realization of the weak shared coin protocol is performed. An invocation of such a protocol is denoted by $\text{GETCOIN}_\delta(b, i)$, where δ is a positive real that represents the agreement parameter of the weak shared coin (see [3]).

First, we prove that the protocol in Figure 1 is correct and efficient. Later we show how to implement the weak shared coin.

Lemma 1. *If some process decides v at time t, then, before time t some process started executing propose(v).*

Proof. The proof is exactly the same of that of Lemma 1 in [9].

Lemma 2. *No two processes decide different values.*

Proof. The proof is exactly the same of that of case (3) of Lemma 4 in [9].

Lemma 3. *Suppose that the following conditions hold:*

i) $\text{MARK}[b, i] =$ `true` *at time t,*
ii) $\text{MARK}[1-b, i] =$ `false` *before time t,*
iii) $\text{MARK}[1-b, i]$ *is set* `true` *at time t' ($t' > t$), and*
iv) *every invocation of both* $\text{GETCOIN}_\delta(b, i)$ *and* $\text{GETCOIN}_\delta(1-b, i)$ *yields value b.*

Then, no process sets $\text{MARK}[1-b, i+1]$ *to* `true`.

Proof. The proof is essentially the same of that of the Claim included in the proof of Lemma 6 in [9].

The next lemma is the heart of the new proof. The difficulty of course is that now we are using protocol $\text{GETCOIN}_\delta(,)$ instead of the "global coins" of [9], and have to contend with the strong adversary. The crucial observation is that if

```
{Initialization}
MARK[0, 0], MARK[1, 0] ← true

{Algorithm for process p}

function propose(v): returns 0 or 1
estimate_p ← v
position_p ← 1
repeat
    MARK[estimate_p, position_p] ← true
    if MARK[1 − estimate_p, position_p + 1]
        newEstimate_p ← 1 − estimate_p
    else if MARK[1 − estimate_p, position_p]
        newEstimate_p ← GETCOIN_δ(estimate_p, position_p)
    else if MARK[1 − estimate_p, position_p − 1]
        newEstimate_p ← estimate_p
    else return(estimate_p) {Decide estimate_p}
    if not MARK[estimate_p, position_p + 1]
        estimate_p ← newEstimate_p
    position_p ← position_p + 1
end repeat
```

Fig. 1. n-process binary consensus for symmetric memory

two teams are in the same position i and the adversary wants to preserve parity between them, it must allow both teams to raise their flags "simultaneously," i.e. at least one teammate in each team must observe parity in the row of flags. But then each team will proceed to invoke $\text{GETCOIN}_\delta(,)$, whose unknown outcome is unfavourable to the adversary with probability at least $(\delta/2)^2$.

Lemma 4. *If* $\text{MARK}[b, i] =$ `true` *at time t and* $\text{MARK}[1-b, i] =$ `false` *before time t, then with probability at least $\delta^2/4$,* $\text{MARK}[1-b, i+1]$ *is always* `false`*.*

Proof. If $\text{MARK}[1-b, i]$ is always `false`, then it can be shown that $\text{MARK}[1-b, i+1]$ is always `false` (the proof is the same of that of Lemma 2 in [9]). So, assume that $\text{MARK}[1-b, i]$ is set to `true` at some time t' (clearly, $t' > t$). Since no invocation of both $\text{GETCOIN}_\delta(b, i)$ and $\text{GETCOIN}_\delta(1-b, i)$ is made before time t, the values yielded by these invocations are independent of the schedule until time t. Thus, with probability at least $\delta^2/4$, all the invocations of $\text{GETCOIN}_\delta(b, i)$ and $\text{GETCOIN}_\delta(1-b, i)$ yield the same value b. From Lemma 3, it follows that, with probability at least $\delta^2/4$, $\text{MARK}[1-b, i+1]$ is always `false`.

Theorem 1. *The protocol of Figure 1 is a randomized solution to n-process binary consensus. Assuming that each invocation of* $\text{GETCOIN}_\delta(,)$ *costs one unit of time, the expected running time per process $O(1)$. Furthermore, with high probability every process will invoke* $\text{GETCOIN}_\delta(,)$ $O(\log n)$ *many times.*

Proof. **(Sketch)** From Lemma 2 the protocol is consistent and from Lemma 1 it is also valid. Thus, the protocol is correct.

As regarding the expected decision time for any process, let $P(i)$ denote the probability that there is a value $b \in \{0, 1\}$ such that $\text{MARK}[b, i]$ is always `false`. From Lemma 4, it follows that

$$P(i) \geq 1 - (1 - \delta^2/4)^{i-1} \qquad i \geq 1$$

Also, if $\text{MARK}[b, i]$ is always `false`, it is easy to see that all the processes decide within $i+1$ iterations of the **repeat** loop. Thus, with probability at least $1-(1-\delta^2/4)^{i-1}$, all the processes decide within $i+1$ iterations of the **repeat** loop. This implies that the expected running time per process is $\text{O}(1)$. The high probability claim follows from the observation that pessimistically the process describing the invocations of $\text{GETCOIN}_\delta(,)$ can be modelled as a geometric distribution with parameter $p := (\delta/2)^2$.

We now come to the implementation of the weak shared coin for symmetric memory, which we accomplish via a slight modification of the protocol of Aspnes and Herlihy [3]. In that protocol the n processes cooperatively simulate a random walk with absorbing barriers. To keep track of the pebble a `distributed counter` is employed. The distributed counter is implemented with an array of n registers, with position i privately owned by process i (that is, naming or asymmetric memory is assumed). When process i wants to move the pebble it updates atomically its own private register by incrementing or decrementing it by one. The private register also records another piece of information namely, the number of times that the owner updated it (this allows one to show that the implementation of the read is linearizable). On the other hand, reading the position of the pebble is a non-atomic operation. To read the counter the process scans the array of registers twice; if the two scans yield identical values the read is completed, otherwise two more scans are performed, and so on. As shown in [3], the expected number of elementary operations (read's and write's) performed by each process is $O(n^4)$.

Since in our setting we cannot use single-writer registers, we use an array C[] of n^2 multiple-writer multiple-reader registers for the counter. The algorithm for a process p is as follows. Firstly, p chooses uniformly at random one of the n^2 registers of C[], let it be the kth. Then, the process proceeds with the protocol of Aspnes and Herlihy by using C[k] as its own register and by applying the counting operations to all the registers of C[]. Snce we are using n^2 registers instead of n, the expected number of steps that each process performs to simulate the protocol is $O(n^5)$. The agreement parameter of the protocol is set to $2e\delta$. Since the expected number of rounds of the original protocol is $\mathrm{O}(n^4)$, by Markov's Inequality, there is a constant B such that, with probability at least $1/2$, the protocol terminates within Bn^5 rounds. It is easy to see that if no two processes choose the same register, then the protocol implements a weak shared coin with the same agreement parameter of the original protocol in $O(n^5)$ many steps. To ensure that our protocol will terminate in any case, if after Bn^5 steps the process has not yet decided then it flips a coin and decides accordingly. Thus, even in case of collision the protocol terminates with a value 0 or 1 within $O(n^5)$ steps. The probability that no two processes choose the same register is

$$\left(1-\frac{1}{n^2}\right)\left(1-\frac{2}{n^2}\right)\cdots\left(1-\frac{n-1}{n^2}\right) \geq \frac{1}{e}.$$

Thus, the agreement parameter of our protocol is at least $1/2 \cdot 1/e \cdot 2e\delta = \delta$. We have proved the following fact.

Proposition 1. *For any $\delta > 0$, a weak shared coin with agreement parameter δ can be implemented in the symmetric model (with randomization) in $O(n^5)$ steps, even against the strong adversary.*

Therefore the expectd running time per process of the protocol of Theorem 1 is the same $O(n^5)$.

In contrast, no protocol exists in the symmetric model for Naming, even assuming the availability of consensus objects and the weak adversary.

Proposition 2. *Suppose that an asynchronous, shared memory machine is such that:*

- *the memory is symmetric;*
- *every process has access to a source of independent, unbiased random bits, and*
- *consensus objects are available.*

Then, still, Naming is impossible even against a weak adversary.

Proof. **(Sketch)** By contradiction suppose there exist such a protocol. Consider two processes P and Q and let only Q go. Since the protocol is wait-free there exists a sequence of steps $\sigma = s_1 s_2 \dots s_n$ taken by Q such that Q decides on a name k_σ. The memory goes through a sequence of states $m_0 m_1 \dots m_n$. The sequence σ has a certain probability $p_\sigma = p_1 p_2 \dots p_n$ of being executed by Q. Start the system again, this time making both P and Q move, but one step at a time alternating between P and Q. With probability p_1^2 both P and Q will make the same step s_1. A simple case analysis performed on the atomic operations (read, write, invoke consensus) shows that thereafter P and Q are in the same state and the shared memory is in the same state m_1 in which it was when Q executed s_1 alone. This happens with probability p_1^2. With probability p_2^2, if P and Q make one more step each, we reach a situation in which P and Q are in the same state and the memory state is m_2. And so on, until, with probability p_σ^2 both P and Q decide on the same identifier, a contradiction.

Thus, naming is "harder" than consensus. The next fact shows that naming is a necessary assumption in Herlihy's universality construction. The proof is omitted from this extended abstract.

Proposition 3. *If processes are identical, deterministic i/o automata without identifiers then Naming is impossible, even if memory is asymmetric and the adversary is weak.*

3 Randomization $\rightarrow$ (Naming + Symmetry = Asymmetry)

In this section we exhibit a protocol to show the non-trivial part of the above "equation" namely, that asymmetric memory is enough for naming, provided randomization is available.

Let us start by informally describing protocol `squeeze` whose task is to assign a unique identifier to each one out of a set of n processes, using a name space of size n. For now, let us assume the availability of objects `selectWinner(i)` with the following semantics. The object can be invoked by a process p with a parameter i; the object response is to return the value *"You own key i!"* to exactly one of the invoking processes, and *"Sorry, look for another key"* to all remaining processes. The choice of the "winner" is non-deterministic. Later we will show that `selectWinner` can be implemented efficiently in a wait-free manner in our setting. With `selectWinner` a naming protocol can be easily obtained as follows: Try each key one by one, in sequence, each time invoking `selectWinner`. This is shown in Figure 2. This protocol always works but linearly many processes always perform linearly many invocations of `selectWinner`, something which could be quite expensive. Thus, we turn our attention to protocol `squeeze` which, with high probability, will only perform $O(\log^2 n)$ such invocations.

Let us then turn our attention to protocol `squeeze`. The name space is divided into *segments*, defined by the following recurrence, where p is a parameter to be fixed later:

$$s_k = p(1-p)^{k-1}n$$

s_ℓ is the last value s_i such that $s_i \geq c\log^2 n$ (c a parameter to be set by the user; the bigger the c the higher the probability that `selectWinner` will be invoked only $O(\log^2 n)$ many times). The first segment consists of the key interval $I_0 := [0, s_1)$; the second segment consists of the key interval $I_1 := [s_1, s_1 + s_2)$; the third of the key interval $I_2 := [s_1 + s_2, s_1 + s_2 + s_3)$, and so on. The final segment I_ℓ consists of the last $n - \sum_{j=1}^{\ell} s_j$ keys. In the protocol, each process p starts by selecting a *tentative key* i uniformly at random in I_0. Then, it invokes `selectWinner`(i); if p "wins," the key becomes final and p stops; otherwise, p selects a second tentative key j uniformly at random in I_1. Again, `selectWinner`(j) is invoked and if p "wins" j becomes final and p stops, otherwise p continues in this fashion until I_ℓ is reached. The keys of I_ℓ are tried one by one in sequence. If at the end p has no key yet, it will execute the protocol `easyButExpensive` of Figure 2 as a back-up procedure. The resulting protocol appears in Figure 3. We will show that: (a) with high probability every process receives a unique key before the back-up procedure, and (b) with probability 1 every process receives a key by the end of the back-up procedure.

The intuition behind `squeeze` is this. On the one hand, if the segments are small enough then, with high probability, for every i, `selectWinner(i)` will be invoked by some process and, consequently, every key i will be assigned (all processes currently looking for a key are "squeezed" in an interval). In turn, this implies that no p will execute the backup procedure. On the other hand, if the segments are big enough , the number of intervals I_i, corresponding to the number of attempts before the backup procedure, will be small. Let us now quantify this intuition. Let p_i be defined by the following recurrence

$$p_i := (1-p)^{i-1}n.$$

If there are no crashes, the number of processes which perform the i-th attempt, i.e. those processes which select a tentative key in I_i, is obviously at least p_i. We want to show that, with high probability, it is at most p_i, for $i < \ell$. If we can show this we are done because then $p_\ell = s_\ell$ and the protocol ensures that every one of the remaining p_ℓ process will receive one of the last s_ℓ keys. A key k is *claimed* if `selectWinner(k)` is invoked by some process. Suppose first that there are no crashes. Then,

$$\Pr[\exists k \in I_i,\, k \text{ not claimed}] \leq \left(1 - \frac{1}{s_i}\right)^{p_i} \leq \exp\left\{-\frac{s_i}{p_i}\right\} \leq \exp\left\{-\frac{1}{p}\right\} = \frac{1}{n^c}$$

for any fixed $c > 0$, provided that

$$p := \frac{1}{k\log n}.$$

With this value of p the number of segments, i.e. ℓ, is $O(\log^2 n)$. The expected running time is therefore

$$T(n) = O(\log^2 n)(1 - n^{-k+2}) + O(n)n^{-k+2} = O(\log^2 n)$$

for $k > 3$. It can be shown that the running time is the same order of the expectation with high probability.

We now give an informal argument showing that when there are crashes the situation can only improve, deferring a formal justification to the full paper. A run of the protocol can be modelled in an equivalent fashion as follows. We have n bins, one for each key, and n white balls, one for each process. As for the key space, the n bins are organized into segments I_i. A process selects a key at random from I_i by throwing a ball in the corresponding bin interval. A key k is claimed if bin k is hit by at least one ball. A run without crashes corresponds to the following process. The n balls are thrown independently at random into the first segment I_1; for each bin which is hit a ball is selected and discarded. The remaining balls are thrown independently at random into I_2, and so on. Let us denote by p_i the number of bins from I_i which are hit. A run with crashes is modelled as follows. When the adversary crashes process p the corresponding ball is painted red. In our balls-and-bins experiment we keep throwing all balls, red and white, but when a bin is hit we always discard a white ball. If we denote by w_i the number of processes which perform the ith attempt and by r_i the number of processes crashed before their ith attempt, then $p_i \leq r_i + w_i$. This is because for each ball which is discarded in a run without crashes we can always find a corresponding (unique) ball in the other run which corresponding to a crashed process or to a process who obtained a key. Therefore, with high probability, at the end each process is either crashed or it has received a unique key.

We now show how to implement `selectWinner` in a wait-free manner in polynomial time. We will assume the availability of objects of type `consensus(i,b)` where $1 \leq i \leq n$ and $b \in \{0, 1\}$. Each invoking process p will perform the invocation using the two parameters i and b; the object response will be the same to all processes and will be *"The consensus value for i is v"* where v is one of the bits b which were proposed. This can be assumed without loss of generality since consensus can be implemented in a wait-free manner in the asymmetric model. Using consensus objects will simplify the presentation. The protocol for `selectWinner`, shown in Figure 4, is as follows. Each process p generates a bit b_1^p at random and invokes `consensus(1,`b_1^p`)`. Let v_1 be the response of the consensus object. If $b_1^p \neq v_1$ then p is a *loser* and exits the protocol. Otherwise, p is still in the game. Now the problem is to ascertain whether p is alone, in which case it is the *winner*, or if there are other processes still in the game. To this end, each remaining process scans the array $W[1, i]$, for $1 \leq i \leq n$, which is initialized to all 0's. If $W[1, i]$ contains a 1 then p declares itself a *loser* and exits; otherwise it writes a 1 in its private position $W[1, p]$ and scans $W[1, -]$ again. If $W[1, -]$ contains a single 1, namely $W[1, p]$ then p declares itself the *winner* and grabs the key, otherwise it continues the game that is, it generates a second bit b_2^p at random, invokes `consensus(2,`b_2^p`)`, and so on. The following facts establish the correctness of the protocol. Their proof is omitted from this extended abstract.

Proposition 4. *If p declares itself the winner then it is the only process to do so.*

Proposition 5. *There is always a process which declares itself the winner. Moreover, with probability $1 - o(1)$ every process p generates $O(\log n)$ many random bits b_i^p.*

These facts establish the correctness of the protocol and the high probability bound on the running time, since consensus can be implemented in the asymmetric model in polynomial time.

Remark 1: Protocol `squeeze` is also a good renaming protocol. Instead of the random bits, each process can use the bits of its own IDs starting, say, from the left hand side. Since the ID's are all different the above scheme will always select a unique winner within $O(|ID|)$ invocation of consensus.

Remark 2: The only part of protocol `squeeze` that actually uses the memory is protocol `selectWinner`. In view of Fact 2 this task must be impossible with symmetric memory, even if randomness and consensus are available. Thus, this is another task for which, strictly speaking, Herlihy's result does not hold and it is another example of something that cannot be accomplished by the power of randomization alone.

References

1. J. Aspnes, Time- and space-efficient randomized consensus. Journal of Algorithms 14(3):414-431, May 1993.
2. J. Aspnes, Lower bounds for distributed coin-flipping and randomized consensus. Journal of the Association for Computing Machinery 45(3):415-450, May 1998.
3. J. Aspnes and M.Herlihy, Fast randomized consensus using shared memory, Journal of Algorithms 11(3):441-461, September 1990.
4. J. Aspnes and O.Waarts, Randomized consensus in $O(n \log n)$ operations per processor, SIAM Journal on Computing 25(5):1024-1044, October 1996.
5. A. Attiya, Gorbach, S. Moran, Computing in totally anonymous shared memory systems, DISC 98, LNCS 1499, pp. 49–61
6. Y. Aumann, Efficient Asynchronous Consensus with the Weak Adversary Scheduler, in Proceedings of the 16th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 1997), pp. 209-218
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. T. D. Chandra, Polylog Randomized Wait-Free Consensus, in Proceedings of the 15th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 1996)
10. T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving Consensus. Journal of the ACM, 43(4):685-722, July 1996.
11. T. D. Chandra and S. Toueg, Unreliable failure detectors for reliable distributed systems. Journal of the ACM, 43(2):225-267, March 1996.
12. M. Herlihy, Wait-Free Synchronization, preliminary version in Proceedings of the 7th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 1988)
13. 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.
14. P. Jayanti, Robust wait-free hierarchies. Journal of the ACM, 44(4):592-614, July 1997.
15. P. Jayanti and S. Toueg, Wake-up under read/write atomicity, WDAG 1990, LNCS 486, pp. 277–288.
16. S. Kutten, R. Ostrovsky and B. Patt-Shamir. The Las-Vegas Processor Identity Problem (How and When to Be Uniique), Proceedings of the 1st Israel Symposium on Theory of Computing and Systems, 1993.
17. R.J. Lipton and A. Park, Solving the processor identity problem in $O(n)$ space, *Inform. Process. Lett.*, 36(1990), 91–94.
18. Wai-Kau Lo and V. Hadzilacos. Using Failure Detectors to Solve Consensus in Asynchronous Shared-Memory Systems, Proceedings of the 8th International Workshop on Distributed Algorithms. Terschelling, The Netherlands, September–October 1994, pp. 280–295.
19. A. Panconesi, M. Papatriantafilou, P. Tsigas and P. Vitanyi, Randomized naming using wait-free shared variables. Distributed Computing (1998) 11:113–124
20. A. Pogosyants, R. Segala and Nancy Lynch, Verification of the Randomized Consensus Algorithm of Aspnes and Herlihy: a Case Study, MIT Technical Memo number MIT/LCS/TM-555, June 1997.

```
protocol simpleButExpensive(): key;

begin
  for k := 1 to n do
    if selectWinner(k) = ''You own key k!'' then return(k);
end
```

Fig. 2. Simple but expensive protocol for Naming

```
protocol squeeze(): key;

begin
  for i := 1 to ℓ do begin
    k := random key in interval I_i;
    if selectWinner(k) = ``You own key k!'' then return(k);
    end;
  for k := n - s_ℓ to n do         {try key in I_ℓ one by one}
    if selectWinner(k) = ``You own key k!'' then return(k);
  return(simpleButExpensive())          {back up procedure}
end
```

Fig. 3. Protocol squeeze

```
protocol selectWinner(i: key);

myBit := ``private bit of executing process'';
attempt := 1;
repeat
  b := random bit;
  if (b = consensus(i, b)) then begin
    scan W[attempt,j] for 1 ≤ j ≤ n;
    if (W[attempt,j] = 0, for all j) then begin
      W[attempt,myBit] := 1;
      scan W[attempt,j] for 1 ≤ j ≤ n;
      if (W[attempt,j] = 0, for all j <> myBit) then return(i); {key is grabbed!}
      else attempt := attempt + 1;                              {keep trying}
    else return(``Sorry, look for another key.'');
end repeat
```

Fig. 4. Protocol selectWinner