Distrib. Comput. (2005)
Digital Object Identifier (DOI) 10.1007/s00446-005-0121-z

On the importance of having an identity or, is consensus really universal?

Harry Buhrman[1], **Alessandro Panconesi**[2], **Riccardo Silvestri**[2], **Paul Vitanyi**[1]

[1] CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
[2] Dipartimento di Informatica, Unversità La Sapienza di Roma, via Salaria 113, 00198 Roma, Italy

Received: 24 January 2002 / Revised: 15 October 2002, 10 September 2004 / Accepted: 19 November 2004
Published online: 10 March 2005 – © Springer-Verlag 2005

Abstract. We show that Naming – the existence of distinct IDs known to all – is a hidden, but 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 relevant differences existing between popular shared memory models.

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 [13]. Roughly speaking, 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 fault-tolerant manner? The notion of fault-tolerance adopted here is that of wait-freedom, i.e. $(n-1)$-resiliency [13]. This question is the starting point of an interesting theory leading to many results and further intriguing questions (see [13,15] among others). One of the basic results of this theory, already contained in the original article of Herlihy, can be stated, somewhat loosely, as follows: If an abstract data type X, together with a shared memory, is powerful enough to implement Consensus for n processes in a fault-tolerant manner then, X, together with a shared memory, is also powerful enough to implement in a fault-tolerant 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 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.

In the **Consensus** problem we are given a set of n asynchronous processes that, as far as this paper is concerned, communicate via a shared-memory. Every process has its own input bit and is to produce its own output bit. Processes can suffer from **crash failures**. The problem is to devise a protocol that can withstand up to $(n-1)$ crash failures, i.e. a **wait-free** protocol, satisfying the following conditions:

- Every non-faulty process terminates;
- All output bits have the same value and,
- The output bit of each processor is the input bit of some process.

The **Naming** problem on the other hand, is as follows: Devise a protocol for a set of n asynchronous 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.

In some sense, this paper is about the relative complexity of Naming to Consensus, and viceversa. We shall mostly concern ourselves with **probabilistic protocols**. Every process in the system, modeled as an i/o automaton, has access to its own source of unbiased random bits. The processes are **asynchronous** and communicate via a **shared memory**. The availability of objects of abstract data type `consensus` and `naming` is assumed. An object of type `consensus` is a subroutine with input parameter $b \in \{0,1\}$. When it is invoked by a process p a bit b' is returned. This bit is the same to all invoking processes and is equal to some of the input bits, i.e. if b' is returned some process p must have invoked the object with input parameter b'. An object of type `naming` is a subroutine without input parameters that, when invoked by a process p, returns a value $v_p \in \{1,..,n\}$, n being the overall number of processes. For any two processes $p \neq q$ we have that $v_p \neq v_q$.

The protocols we devise should be wait-free in spite of the **adversary**, the "malicious" non-deterministic scheduling agent 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 history of the protocol execution so far. This is the so-called *adaptive* adversary. In contrast, sometimes it is assumed that the adversary decides the entire execution schedule beforehand. This weaker foe is the so-called *oblivious* adversary.

In the literature two shared-memory models are widespread. The first assumes **multiple reader - multiple writer** registers. In this model each location of the shared memory can be written and 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. 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.

We are now ready to state the results of this paper. Let us start by restating Herlihy's universality result.

Theorem. [Herlihy] *Suppose that n asynchronous processes interact via a shared memory and that,*

(i) *Memory registers are multiple reader–multiple writer;*
(ii) *Each process has its own unique identifier;*
(iii) *Objects of type* `consensus` *are available to the processes.*

Then, any abstract data type T can be implemented in a wait-free manner for the n processes.

The first question we consider in this paper is: What happens if the second hypothesis is removed? Can distinct identifiers be generated from scratch in this memory model? The answer is negative *even assuming the availability of Consensus objects.*

Proposition 1 [Naming is impossible] *Suppose that n asynchronous processes* ***without identifiers*** *interact via a shared memory and that,*

(i) *Memory registers are multiple reader–multiple writer;*
(ii) *Each process has access to its own source of unbiased random-bits;*
(iii) *Objects of type* `consensus` *are available to the processes;*
(iv) *The adversary is oblivious.*

Yet, wait-free Las Vegas Naming is impossible.

This result is simple to prove, but it is interesting in several respects. First, Consensus by itself is **not** universal, for wait-free Naming objects cannot be implemented in a wait-free manner with Consensus alone. Also, note that the result holds even with randomization, a powerful "symmetry–breaker", and against the weak oblivious adversary. In particular, it holds for the memory model used in Herlihy's universal construction.

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. Note that Montecarlo Naming is trivial – each process generates $O(\log n)$ many random bits and with probability $1 - o(1)$ no two of them will be identical. Therefore, at least at the outset, only the question of the existence of Las Vegas protocols is of interest.

Proposition 1 shows that the power of randomization to "break the symmetry"is limited. If we start from a completely symmetric situation, it is impossible to generate identifiers that are surely distinct. In stark contrast with the previous result, as we prove in this paper, the following holds.

Theorem 1 [Consensus is easy] *Suppose that n asynchronous processes* ***without identifiers*** *interact via a shared memory and that,*

(i) *Memory registers are multiple reader – multiple writer;*
(ii) *Each process has access to its own source of unbiased random-bits;*
(iii) *The adversary is adaptive.*

Then, there exist Las Vegas, wait-free Consensus protocols for n processes whose complexity is polynomial in expectation and with high probability.

Notice that while Proposition 1 establishes the impossibility of Naming even against the oblivious adversary, here the adversary is adaptive. We remark that the protocol can conceivably use super-polynomial space and time, even though the probability that this happens is inverse polynomial.

Incidentally, Theorem 1 shows that hypothesis (iii) of Proposition 1 is superfluous, for Consensus objects can be simulated via software in a wait-free manner. It is well-known that hypothesis (ii) is necessary, even if the adversary is oblivious (see, for instance, [20,6]).

In some sense Naming captures the notion of complete asymmetry among the processes. If we start from a completely symmetric situation it embodies the intuitive notion of complete break of symmetry. Proposition 1 shows that not even randomness can break the symmetry to such an extent. This leads to the question of "how much" asymmetry is needed for Naming. A memory consisting of multiple reader – single writer registers has some degree of built-in asymmetry. Indeed, if the absolute address of the registers could be known Naming could be trivially solved by ranking the physical addresses. Therefore in Sect. 3 we introduce a memory model whose registers are multiple reader – single writer but for which it is impossible to obtain the physical addresses. For clarity of exposition we shall call these *anonymous multiple reader – single writer* registers.

Although our motivation is mainly theoretical, similar models have been used to study certain situations in large dynamically changing systems where a consistent indexing scheme is difficult or impossible to maintain [18]. Moreover this model could make sense in cryptographic systems where this kind of consistency is to be avoided.

We show the following. Assume that the memory consisting of anonymous multiple reader – single writer registers is initialized*fairly* that is, all registers are initially 0 (or any other fixed value). Moreover, processes do not have ID's. Then,

- Naming is impossible for deterministic processes;
- The size of the memory is a lower-bound for any wait-free Naming protocol. That is, if m is the memory size, any wait-free Naming protocol must take $\Omega(m)$ time;
- There exists a Las Vegas, wait-free Naming protocol for n processes whose running time is polynomial in expectation. The protocol can use super-polynomail time and space, but that happens with inverse polynomial probability. Furthermore the key space from which identifiers are drawn has size n, which is optimal. The protocol can withstand the adaptive adversary.

Thus, by themselves, neither randomization nor multiple reader – single writer registers can break the symmetry. What is needed is their acting together.

To summarize, with randomization, multiple reader – single writer registers are inherently symmetry-breaking, whereas Consensus is not.

The last result improves on previous work in [21] in which Naming protocols with almost-optimal key range are given. In fact, we prove two versions of the third result above. We first give a simple protocol whose running time is $\Theta(C\ n^2 \log n)$ w.h.p., where C is the time required for a call of an object of type `consensus`. We also give a faster protocol named `squeeze` whose expected running time is $O(C\ n \log^3 n)$.

As a by-product we also show that an object we call `selectWinner` cannot be implemented by Consensus alone, i.e. without Naming, even if randomization is available and the adversary is oblivious. The semantics of `selectWinner` is the following: the object selects a unique winner among the invoking processes. After the invocation each process knows if it is a winner or a loser. In essence, `selectWinner` is a `test-and-set` without reset.

Since any deterministic protocol must use a key range of size at least $2n - 1$ in order to be wait-free [14], this is yet another instance in which randomization is proven to be more powerful than determinism, as far as fault-tolerant computing is concerned.

Our results show, perhaps surprisingly, that multiple reader–single writer registers are more powerful than multiple reader–multiple writer registers, even though the latter might represent a faster alternative. This highlights an important difference between the two models.

Our Theorem 1 is obtained by combining several known ideas and protocols, in particular those in [3] and [10]. When compared to the protocol in [3] it is, we believe, simpler, and its correctness is easier to establish (see, for instance, [22]). Moreover, it works in the more "symmetric" multiple reader – multiple writer memory model and can deal with the adaptive adversary. In contrast, the protocol in [10] can only withstand the "intermediate" adversary, whose power lies somewhere between the more traditional oblivious and adaptive adversaries we consider. From the technical point of view, our Propositions 1 and 3 use ideas contained in [17] to which we refer for other interesting related results (those results however are not directly relevant to the issue under consideration here, namely the power of Consensus). Other related work can be found in [5,16].

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 shared memory consists of multiple reader – multiple writer registers, (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. Ours is a modification of the protocol proposed by Chandra [10] to withstand the intermediate adversary. 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 multiple reader – single writer registers. 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 super-polynomial space could be needed. We leave it as an open problem whether this is necessary. In the sequel we will make use of the notion of weak shared coin of [3]. Basically a weak-shared coin protocol returns Head to all invoking processes with some probability p; it returns Tail to all processes with the same probability p; and it returns 0 to some processes and 1 to the remaining ones with probability $1 - 2p$. Although our treatment below is self-contained, we refer to [3] for more details.

The protocol, shown in Fig. 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 $\textsc{Mark}[b, \cdot]$– which are to be raised one after the other from left to right by the members of team b, cooperatively and asynchronously. The variable $position_p$ of each process p 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 process p is defined by the variable $myTeam_p$. 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). For instance, if p's team corresponds to the value b then, $\textsc{Mark}[b, position_p]$ is set to true. Thus, as far as process p is concerned, the value of its own team counter is $position_p$ (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 $tentativeNewTeam_p$ 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 $\textsc{GetCoin}_\delta(\cdot, \cdot)$ and sets $tentativeNewTeam_p$ 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. The setting of $tentativeNewTeam_p$ is, as the name suggests, tentative. 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 its old team and continues; otherwise, it does join the team specified by $tentativeNewTeam_p$. The array $\textsc{Mark}[\cdot, \cdot]$ is implemented with `multiple reader - multiple writer` registers, while the other variables are local to each process and accessible to it only. The local

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

{Algorithm for process p}

function propose(v): returns 0 or 1
1.    $myTeam_p \leftarrow v$; $otherTeam_p \leftarrow 1 - myTeam_p$
2.    $position_p \leftarrow 1$
3.    repeat
4.        MARK[$myTeam_p, position_p$] ← true
5.        if MARK[$otherTeam_p, position_p + 1$]
6.            $tentativeNewTeam_p \leftarrow 1 - myTeam_p$
7.        else if MARK[$otherTeam_p, position_p$]
8.            $tentativeNewTeam_p$
                  ← GETCOIN$_\delta$($myTeam_p, position_p$)
9.        else if MARK[$otherTeam_p, position_p - 1$]
10.           $tentativeNewTeam_p \leftarrow myTeam_p$
11.       else return($myTeam_p$) {Decide $myTeam_p$}
12.       if not MARK[$myTeam_p, position_p + 1$]
13.           $myTeam_p \leftarrow tentativeNewTeam_p$
14.       $position_p \leftarrow position_p + 1$
      end repeat
```

Fig. 1. n-process binary Consensus for multiple reader – multiple writer registers

variables can assume only a finite (constant) set of values and can therefore be "hardwired" in the states of the i/o automaton representing the process.

The only, but crucial, difference between our protocol and that of Chandra concerns the procedure GETCOIN$_\delta(\cdot, \cdot)$. In Chandra's setting essentially it is possible to implement "via software" a *global coin*, thanks to the Naming assumption and the special assumption concerning the power of the adversary ("intermediate" instead of adaptive). In the implementation in Fig. 1, we use a protocol for a weak shared coin for multiple reader – multiple writer registers. 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 GETCOIN$_\delta(b, i)$, where δ is a positive real that represents the agreement parameter of the weak shared coin (see [3]). GETCOIN$_\delta(b, i)$ satisfies the following conditions. Upon invocations with values b and i, it returns 0 to all invoking processes with probability $p \geq \delta/2$; it returns 1 to all invoking processes with probability $p \geq \delta/2$; and, it returns 0 to some and 1 to the others with probability at most $1 - \delta$ [3].

First, we prove that the protocol in Fig. 1 is correct and then analyze its complexity. 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 [10]. □

Lemma 2 *No two processes decide different values.*

Proof The proof is exactly the same as that of case (3) of Lemma 4 in [10]. □

Lemma 3 *Suppose that the following conditions hold:*

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

Then, no process sets 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 [10]. □

The next lemma is the heart of the new proof. The difficulty of course is that now we are using protocol GETCOIN$_\delta(\cdot, \cdot)$ instead of the "global coins" of [10], and have to contend with the adaptive adversary instead of the intermediate one. The crucial observation is that if 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 member in each team must observe parity in the row of flags. But then each team will proceed to invoke GETCOIN$_\delta(\cdot, \cdot)$, whose unknown outcome is unfavorable to the adversary with probability at least $(\delta/2)^2$.

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

Proof If MARK$[1 - b, i]$ is always `false`, then it can be shown that MARK$[1 - b, i + 1]$ is always `false` (the proof is the same of that of Lemma 2 in [10]). So, assume that MARK$[1 - b, i]$ is set to `true` at some time t' (clearly, $t' > t$). Since no invocation of both GETCOIN$_\delta(b, i)$ and 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 GETCOIN$_\delta(b, i)$ and GETCOIN$_\delta(1 - b, i)$ yield the same value b. From Lemma 3, it follows that, with probability at least $\delta^2/4$, MARK$[1 - b, i + 1]$ is always `false`. □

Theorem 2 *The protocol of Fig. 1 is a randomized solution to n-process binary Consensus. Assuming that each invocation of* GETCOIN$_\delta(\cdot, \cdot)$ *costs one unit of time, the expected running time per process $O(1)$. Furthermore, with high probability every process will invoke* GETCOIN$_\delta(\cdot, \cdot)$ $O(\log n)$ *many times.*

Proof From Lemma 2, if any two processes decide, they decide on the same value. From Lemma 1 we know that the decision value is the input bit of some process. We now show that all processes decide within a finite number of steps and that this number is polynomial both in expectation and with high probability.

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 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 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 O(1). The high probability claim follows from the observation that pessimistically the process describing the invocations of $\text{GETCOIN}_\delta(\cdot,\cdot)$ can be modeled as a geometric distribution with parameter $p := (\delta/2)^2$. □

We now come to the implementation of the weak shared coin for multiple reader – multiple writer registers, 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 multiple reader – single writer 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). While moving the pebble takes constant time, reading its position is a non-atomic operation. To read the position of the pebble 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. We shall refer to this non-atomic read of the whole array as a *scan*. As shown in [3], the expected running time per process of the protocol is $O(n^3)$ scans. In terms of elementary operations (reads and writes) performed by each process this is $O(n^4)$, since each scan takes O(n).

In our setting we cannot use single-writer registers therefore 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[]. Since we are using n^2 registers instead of n, the expected number of steps that each process performs to simulate the protocol is $O(n^5)$. This is because the expected number of scans per process remains the same, but now each scan takes $\Theta(n^2)$ steps (read's). The agreement parameter of the protocol is set to $2e\delta$. Since the expected number of rounds of the original protocol is 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, in any case the protocol terminates returning a value 0 or 1 to the calling process 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.

Lemma 5 *For any $\delta > 0$, a weak shared coin with agreement parameter δ can be implemented with multiple reader – multiple writer registers (with randomization) in $O(n^5)$ steps, even against the adaptive adversary.*

Corollary 1 *The expected running time per process of the protocol of Fig 1 is $O(n^5)$. The protocol can use super polynomial space and time, but this happens with inverse polynomial probability.*

We show next that, in contrast, no protocol exists for Naming if we use multiple reader – multiple writer registers, even assuming the availability of Consensus objects and the oblivious adversary. The proof uses ideas from [17].

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

- *registers are multiple reader – multiple writer;*
- *every process has access to a source of independent, unbiased random bits, and*
- *Consensus objects are available.*

Then, still, Naming is impossible even against an oblivious adversary.

Proof 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 $\sigma = s_1 s_2 \ldots s_n$ taken by Q such that Q decides on a name k_σ. The memory goes through a sequence of states $m_0 m_1 \ldots m_n$. The sequence σ has a certain probability $p_\sigma = p_1 p_2 \ldots p_n$ 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 a necessary assumption in Herlihy's universality construction.

3 Naming with multiple reader – single writer registers

We now come to the question of whether a memory consisting of multiple reader – single writer registers can be used to break the symmetry. Let us first give an overview of the results we will be discussing.

First, in § 3.1 we will define the memory model formally (anonymous multiple reader – single writer registers) and prove some impossibility results. In particular we show that Naming remains impossible for *deterministic* processes. For randomized processes we prove that memory size is a lower bound on the running time, if we want wait-freedom. Thus randomization is necessary for Naming. To show that it is sufficient, we reduce the Naming problem to another synchronization task: selecting a unique winner out of n competing processes. We introduce an object called `selectWinner(i)` with the following semantics. The object is invoked by any out of n asynchronous processes with a parameter i; the 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. Thus, the object selects a unique *winner* for key i out of n processes. The choice of the winner is non-deterministic. In essence, this is a test-and-set without reset.

In § 3.2 we give a wait-free implementation of `selectWinner` for n randomized processes communicating via multiple reader – single writer registers. Finally, in § 3.3 we show how to reduce the Naming problem to `selectWinner` in a wait-free manner. That is, we give a wait-free implementation of Naming for n randomized processes communicating via multiple reader – single writer registers that have access to objects of type `selectWinner`. Since the latter was shown to admit a wait-free implementation in § 3.2, we have that Naming too admits a wait-free implementation. The running time of our protocols is polynomial both in expectation and with high probability.

3.1 Model definition and impossibility results

Let us define the notion of anonymous multiple reader – single writer registers. Recall that essentially what we are interested in is a memory model in which the physical address of the registers remain hidden from the processes. Here is the model.

A set of n processes communicate by means of a shared memory. Memory registers are `multiple reader - single writer`. While every process is allowed to read the contents of any register, each register is owned by some unique process which is the only process that can write on that register. If this asymmetry were not somehow hidden from the process, the Naming problem would have trivial solutions. Specifically, if the physical addresses of the writable registers of each process were common knowledge, this directly would induce distinct IDs. So, to hide the physical addresses from the processes, we introduce the following formal model. Each process p accesses the m registers by means of a permutation π_p. 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. In particular processes cannot obtain the physical address of the registers. To rule out trivial solutions to the Naming problem it is further assumed that the permutations are chosen so that the set of indices (not the physical addresses) of *writable* registers is the same for all the processors. In case of concurrent access to the same register it is assumed that the adversary "complies with" some non-deterministic, but fair, policy.

Besides registers of the shared memory, each process can have local registers, it is the only process to have access to them, and it can distinguish between a local register and a shared register.

We now show that in this model Naming is impossible for deterministic processes.

Proposition 3 *Suppose that the memory consists of anonymous multiple reader – single writer registers initially set to 0 (or any other fixed value). Then, if processes are identical, deterministic i/o automata without identifiers, Naming is impossible. That holds for any choice of the permutations π.*

Proof Consider n processes $p_1, \dots, p_n$ that are identical deterministic i/o automata without identifiers. The shared memory is anonymous: any process p_k accesses the m registers by means of permutation π_k. That is, when process p_k performs a `Read(`i`)` operation, the result will be the content of the register having physical address π_k^i. Analogously for the `Write(`i`, `v`)` operations. We assume that `Write(`i`, `v`)` is legal only if $i \leq m/n$. That is, the indices of the writable registers are $1, 2, \dots, m/n$. We show the impossibility of Naming even against the very simple adversary with the alternating execution schedule: the execution proceeds in rounds, each round consists of a step for each process $p_1, \dots, p_n$.

We need some notation. We call the map of the contents of all the registers, shared and local, the absolute view. It is a function that maps each physical address a to the content of the register of physical address a. Given an absolute view V remains determined the local views $L_k(V)$ for $k = 1, \dots, n$. The local view $L_k(V)$ is the map of the contents of the local registers of p_k and of the shared registers through the permutation π_k. In other words, $L_k(V)$ is a function that maps each index j to the content of the register having index j w.r.t. the process p_k. In particular, if j is an index of a shared register then the local view maps j to the content of the register of physical address π_k^j. We have already assumed that the set of indices of shared registers is the same for all the processes (i.e. the set $\{1, 2, \dots, m\}$). We further assume that the set of indices of local registers is the same for all the processes. Thus, the entire set of indices is the same for all the processes. Therefore, the domains of all the local views coincide. An instant configuration, or simply a configuration, for process p_k is a pair (L, s) where L is a local view and s is a state of p_k.

Let s_t^k be the state of p_k at the beginning of round t. Let V_t be the absolute view at the beginning of round t. A simple case analysis performed on the atomic operations (read, write, local operations) shows that, for any t, if all the configurations $(L_k(V_t), s_t^k)$ for $k = 1, \dots, n$ are equal then after the execution of round t the resulting configurations $(L_k(V_{t+1}), s_{t+1}^k)$ are equal again. This fact together with the assumption that the initial configurations $(L_k(V_0), s_0^k)$ are equal imply that the processes cannot select unique identifiers. □

Thus randomization is necessary to solve Naming. In the next two subsections we will show that it is sufficient. The next proposition however shows that the size of the memory is a lower bound on the running time.

Proposition 4 *Suppose that an asynchronous, shared memory machine whose multiple reader – single writer registers are anonymous is such that:*

- *all registers are initially set to 0 (or any other fixed value);*
- *every process has access to a source of independent, unbiased random bits, and*
- *Consensus objects are available.*

Then, for any wait-free protocol for the Naming problem there exists a choice of the permutations π that forces the protocol to terminate in $\Omega(m)$ expected time, where m is the size of memory. That holds even if the adversary is oblivious.

Proof Since the protocol is wait-free, without loss of generality, we show the lower bound for the case of only two processes p and q. We prove that for any protocol there is a choice of the

```
protocol selectWinner(i: key): outcome;

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

Fig. 2. Protocol selectWinner

permutations π_p and π_q such that the expected running time is at least $m/2$, where m is the size of memory. By contradiction suppose there exists a protocol whose expected running time is $T < m/2$. Let only q go. Since the protocol is wait-free and its expected running time is bounded by T, there exists a sequence of $t \le T$ steps $\sigma = s_1 s_2 \ldots s_t$ taken by q such that q decides on a name k_σ. For every $i = 1, 2, \ldots, t$, let v_i be the view of the shared memory by q after the execution of step s_i. The sequence σ has a certain probability $p_\sigma = p_1 p_2 \ldots p_t$ of being executed by q. Observe that the sequence σ is a valid execution of the protocol for every possible choice of the permutations π_p and π_q by means of which p and q access the m registers. Since $m > 2t$, we can choose the two permutations so that if both processes were executing the sequence σ then the sets of registers they access are disjoint. 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 processes 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 their view of the shared memory is v_1 the same of q 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 their view of the shared memory is v_2. And so on, until, with probability p_σ^2 both p and q decide on the same identifier, a contradiction. □

3.2 Wait-free implementation of SelectWinner

In this subsection we show how to implement `selectWinner` in a wait-free manner in polynomial time.

By Theorem 2 we can assume without loss of generality the availability of objects of type `consensus(i,b)` where $1 \le i \le n$ 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 that were proposed.

The protocol for `selectWinner`, shown in Fig. 2, 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 p is to ascertain whether it is alone, in which case it is the unique *winner*, or if there are other processes still in the game. To this end, p (and every other process still in the game) scans the array $W[1,i]$, for $1 \le i \le n$, that 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 observations and lemma establish the correctness of the protocol.

Observation 1 *If p declares itself the winner then it is the only process to do so.*

Observation 2 *There is a winner with probability 1.*

Lemma 6 *With probability* $1 - o(1)$*, every process p running the protocol of Fig. 2 generates* $O(\log n)$ *many random bits* b_i^p *and the number of bit operations per process is* $O(C\ n\ \log n)$*, where C is the cost of one invocation of an object of type* `consensus`.

Proof We refer to an iteration of a repeat loop of protocol `selectWinner` as a *round*, i.e. the round i refers to the set of iterations of the repeat loop in which the participating processes toss their private coin for the ith time. We assume pessimistically that the Consensus object of protocol `selectWinner` is under the control of the adaptive adversary, subject to the following rules. Denoting with $1, \ldots, k$ the processes that perform the ith coin toss, If $b_i^1 = b_i^2 = \ldots = b_i^k = 0$ or $b_i^1 = b_i^2 = \ldots = b_i^k = 1$ then the adversary can respond with Consensus value 0 or 1, respectively. Otherwise the adversary can respond with any value.

The goal of the adversary is to maximize the number of rounds. Therefore its best policy is to return the Consensus value that eliminate the smallest number of processes, i.e. the best strategy is to return the majority bit. The probability that the ith outcome of process p is the majority value depends on the number of processes, but it is easily seen to be maximized when there are two processes. The probability that the minority value is the outcome of at least $1/4$ of the processes depends on the number of processes, but it is monotonically decreasing. Therefore the smallest value is $1/4$, when just 2 processes are involved.

We call a run *successful* if the minority value is the outcome of at least $1/4$ of the processes (i.e. at least $1/4$ of the processes are eliminated). Then, $\log_{4/3} n$ many successful rounds suffice to select a winner. A straightforward application of the Chernoff-Hoeffding bounds show that with probability $1 - o(1)$ at least $\log_{4/3} n$ rounds out of $8 \log_{4/3} n$ many will be successful.

Since every iteration of `selectWinner` costs $O(n)$ steps, the claim follows. □

Thus we obtain the following.

Lemma 7 *The protocol shown in Fig. 2 is a wait-free implementation of the objects* `selectWinner(i)` *for n randomized, asynchronous processes that communicate by means of*

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

Fig. 3. Simple but expensive protocol for Naming

multiple reader – single writer registers, even if the adversary is adaptive. The running time per process is $O(C\ n\ \log n)$ bit operations, both in expectation and with high probability, where C is the cost of one invocation of an object of type `consensus`.

3.3 Reducing Naming to Select-Winner

Finally, in this section we show how to reduce Naming to Select-Winner in polynomial time. First we give a simple implementation of Naming and then a more sophisticated and efficient one.

Assuming the availability of objects of type `selectWinner`, it is a simple matter to give a wait-free implementation of Naming: Each of the n processes tries to grab a key. If it succeeds it stops, otherwise the next key is tried. This is done in Fig. 3.

Thus, recalling Lemma 7 we obtain the following.

Proposition 5 *Suppose that n asynchronous processes* ***without identifiers*** *interact via anonymous multiple reader – single writer registers and that,*

(i) *Each process has access to its own source of unbiased random-bits;*
(ii) *The adversary is adaptive.*

Then, protocol `simpleButExpensive` *is a wait-free Las Vegas solution to Naming whose running time is polynomial in expectation and with high probability*

The high probability statement follows from Lemma 6 and Sect. 2 in which it given a wait-free, polynomial-time implementation of objects of type `consensus`. Since our implementation of `consensus` can use super-polynomial space with inverse polynomial probability, the same holds for our implementation of `selectWinner`.

Although the expected running time of `simpleButExpensive` is polynomial, given the high cost of invoking `selectWinner` we turn our attention to protocol `squeeze`. This protocol in expectation will only perform $O(\log^2 n)$ invocations of `selectWinner` instead of linearly many.

In protocol `squeeze` the name space is divided into *segments* I_k of length s_k. In the following definition p is a parameter between 0 and 1 to be fixed later:

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

To simplify the presentation we assume without loss of generality that all s_i's are integral. Let ℓ be the maximal integer i such that $s_i \geq \log^2 n$. The first segment consists of the key interval $I_1 := [0, s_1)$; the second segment consists of the key interval $I_2 := [s_1, s_1 + s_2)$; the third of the key interval $I_3 := [s_1 + s_2, s_1 + s_2 + s_3)$, and so on. Thus each I_k has s_k keys, $1 \leq k \leq \ell$, and the final segment $I_{\ell+1}$ consists of the last $n - \sum_{j=1}^{\ell} s_j = n - n[1-(1-p)^{\ell+1}] = n(1-p)^{\ell+1}$ keys. In the protocol, each process p starts by selecting a tentative key i uniformly at random in I_1. 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_2. Again, `selectWinner(j)` is invoked and if p "wins" j becomes final and p stops, otherwise p continues in this fashion until $I_{\ell+1}$ is reached. The keys of $I_{\ell+1}$ are tried one by one in sequence. If at the end p has no key yet, it will execute the protocol `simpleButExpensive` of Fig. 3 as a back-up procedure. The resulting protocol appears in Fig. 4.

Assuming the availability of objects of type `selectWinner`, protocol `squeeze` assigns a key to every non-faulty process with probability 1. This follows, because the protocol ensures that `selectWinner(i)` is invoked for every i, $1 \leq i \leq n$, and each such invocation assigns a key to exactly one process. We will now argue that with high probability every process receives a unique key before the back-up procedure, and that therefore the number of invocations of `selectWinner` objects is $O(\log^2 n)$ per process w.h.p..

Lemma 8 *The expected number of invocations of* `selectWinner` *per process in protocol* `squeeze` *is $O(\log^2 n)$.*

Protocol `squeeze` maintains the following invariant (w.h.p.). Let P_k be the set of processes that after $k-1$ attempts still have to grab a key. Their k-th attempt will be to select a key at random in segment I_k. Then, $|P_k| \approx |I_k| \log n \gg |I_k|$ (hence the protocol "squeezes" P_k into I_k). Once the numbers are plugged in it follows that, with high probability, every key in I_k will be claimed by some process, and this for all k. Since every key is claimed w.h.p. before the back-up procedure, every process, w.h.p., receives a key within $O(\log^2 n)$ invocations of `selectWinner`. By setting the parameter p appropriately it is possible to keep the number of segments small, i.e. $O(\log^2 n)$, while maintaining the invariant $|P_k| \gg |I_k|$ for each segment.

Let us focus first on a run without crashes. Let p_i be defined by

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

Since each segment I_k can capture at most s_k processes, $|P_i| \geq p_i$. We want to show that, with high probability, $|P_i| = p_i$, 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. Then, since there are no crashes,

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

```
protocol squeeze(): key;

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

Fig. 4. Protocol squeeze

for any fixed $c > 0$, provided that

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

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

$$\begin{aligned} E[T(n)] &= O(\log^2 n)(1 - \ell/n^c) + O(n)\ell/n^c \\ &= O(\log^2 n) \end{aligned}$$

for $c > 3$.

We now argue that with crashes the situation can only improve. Let C_1 be the set of processes that crash before obtaining a response to their invocation of `selectWinner` in I_1. Let F_1 (F as in free) be the set of keys of I_1 that are not claimed after processes in P_1 have made their random choice. If F_1 is non empty, assign processes of C_1 to keys of F_1 in a one-to-one fashion. Let f_1 be this map. Then, the probability that a key is claimed before any process under this new scheme is no lower than in a run without crashes. We then set C_2 to be the set of processes of P_2 that crashed before obtaining a response for their invocation of `selectWinner` in I_2, union $C_1 - f_1(C_1)$. Again, after processes in P_2 randomly select keys in I_2, assign processes of C_2 to keys of F_2 by means of a one-to-one function f_2. Thus, again, the probability that a key in I_2 is claimed is higher than in a run without crashes. And so on. Thus, we have the following.

We have proved the following.

Theorem 3 *Protocol* `squeeze` *is a Las Vegas, wait-free Naming protocol for anonymous multiple reader – single writer registers, whose running time is* $O(S \ \log^2 n)$ *with probability* $1 - o(1)$*, where S is the cost of one invocation of an object of type* `selectWinner`.

By Lemma 7 we obtain the following corollary.

Corollary 2 *Protocol* `squeeze` *is a Las Vegas, wait-free Naming protocol for anonymous multiple reader – single writer registers, whose running time is* $O(C \ n \ \log^3 n)$ *with probability* $1 - o(1)$*, where C is the cost of one invocation of an object of type* `consensus`.

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 Proposition 2 this task must be impossible with multiple reader – multiple writer registers, 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. Aspnes J: Time- and space-efficient randomized consensus. J Algorithms 14(3):414–431 (1993)
2. Aspnes J: Lower bounds for distributed coin-flipping and randomized consensus. J Assoc Comput Machin 45(3):415–450 (1998)
3. Aspnes J, Herlihy M: Fast randomized consensus using shared memory, J Algorithms 11(3):441–461 (1990)
4. Aspnes J, Waarts O: Randomized consensus in $O(n \log n)$ operations per processor. SIAM J Comput 25(5):1024–1044 (1996)
5. Attiya A, Gorbach ♣, Moran S: Computing in totally anonymous shared memory systems, DISC 98. LNCS 1499, pp. 49–61
6. Attiya H, Welch EJ: Distributed Computing, McGraw-Hill
7. Aumann Y: 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
8. Bar-Noy A, Dolev D: 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
9. Borowsky E, Gafni E: Immediate Atomic Snapshots and Fast Renaming. In: Proceedings of the 12th ACM Symposium on Principles of Distributed Computing, 1993, pp. 41–52
10. Chandra TD: Polylog Randomized Wait-Free Consensus. In: Proceedings of the 15th ACM SIGACT-SIGOPS. Symposium on Principles of Distributed Computing (PODC 1996)
11. Chandra TD, Hadzilacos V, Toueg S: The weakest failure detector for solving Consensus. J ACM 43(4):685–722 (1996)
12. Chandra TD, Toueg S: Unreliable failure detectors for reliable distributed systems. J ACM 43(2):225–267 (1996)

13. Herlihy M: Wait-Free Synchronization. Preliminary version in Proceedings of the 7th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 1988)
14. Herlihy M, Shavit N: The Asynchronous Computability Theorem for t-Resilient Tasks. In: Proc. 25th ACM Symp. Theory of Computing, 1993, pp. 111–120
15. Jayanti P: Robust wait-free hierarchies. J ACM 44(4):592–614 (1997)
16. Jayanti P, Toueg S: Wake-up under read/write atomicity, WDAG 1990, LNCS 486, pp. 277–288
17. Kutten S, Ostrovsky R, Patt-Shamir B: The Las-Vegas Processor Identity Problem (How and When to Be Unique). Proceedings of the 1st Israel Symposium on Theory of Computing and Systems, 1993
18. Lipton RJ, Park A: Solving the processor identity problem in $O(n)$ space. Inform Process Lett 36, 91–94 (1990)
19. Wai-Kau Lo, Hadzilacos V: 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
20. Lynch N: Distributed Algorithms, Morgan Kaufmann
21. Panconesi A, Papatriantafilou M, Tsigas P, Vitanyi P: Randomized naming using wait-free shared variables. Distrib Comput 11:113–124 (1998)
22. Pogosyants A, Segala R, Lynch N: Verification of the Randomized Consensus Algorithm of Aspnes and Herlihy: a Case Study. MIT Technical Memo number MIT/LCS/TM-555, June 1997