Anonymous Agreement: The Janus Algorithm

Zohir Bouzid[1,⋆], Pierre Sutra[1,⋆⋆], and Corentin Travers[2,⋆⋆⋆]

[1] University Pierre et Marie Curie - Paris 6, LIP6-CNRS 7606, France
name.surname@lip6.fr
[2] LaBRI University Bordeaux 1
name.surname@labri.fr

Abstract. We consider the consensus problem in an n-process shared-memory distributed system when processes are anonymous, i.e., they have no identities and are programmed identically.

We present Janus, a new anonymous consensus algorithm that reaches decision after $O(\sqrt{n})$ writes in every solo execution. The set of values that can be proposed is unbounded and the algorithm tolerates an arbitrary number of crash failures. The algorithm relies on an anonymous eventual leader election mechanism. Furthermore, during solo executions in which a non-faulty process is elected since the beginning, the individual step complexity of Janus is $O(n)$, matching a recent lower bound by Aspnes and Ellen (SPAA 2011).

The algorithm is then extended to the case of *homonymous* system in which c, $1 \leq c \leq n$, identities are available. In every solo execution, the modified algorithm achieves $O(\sqrt{n-c+1}+\frac{\log c}{\log\log c})$ individual write complexity and $O(n-c+\frac{\log c}{\log\log c})$ individual step complexity.

Keywords: Anonymity, asynchronous shared memory, consensus, failure detectors, homonym processes, indulgent algorithms.

1 Introduction

In a typical distributed system, processes are *eponymous*, i.e., they have unique identities. On the other hand, in *anonymous* systems, processes have no identity and are programmed identically. When provided with the same input, processes in such systems are indistinguishable. Anonymity adds a new, challenging, difficulty to distributed computing.

From a practical point of view, anonymity is sometimes unavoidable. For example, consider a system composed of many tiny nodes, e.g., sensors networks. Sensors nodes might have limited storage and computational capability, and might not have been provided with unique identifiers [2]. Some other systems, like peer-to-peer file sharing applications [13], might require users to remain anonymous as a prerequisite to ensure privacy. See [19] for more details regarding anonymous computing and privacy.

⋆ Supported by DIGITEO project PACTOLE.
⋆⋆ Supported in part by the ANR projects PROSE and CONCORDANT.
⋆⋆⋆ Supported in part by the ANR project DISPLEXITY.

A. Fernández Anta, G. Lipari, and M. Roy (Eds.): OPODIS 2011, LNCS 7109, pp. 175–190, 2011.

Recently, several papers [4,5,14,22,24,29] have addressed the question of the computational power of anonymous systems, with an emphasis on the consensus problem. In particular, Aspnes and Ellen [4] have shown that, when the number of proposed values is unbounded, the solo step complexity of consensus is $\Theta(n)$ in an n-process system. This paper presents a new, efficient, consensus algorithm for anonymous system.

The consensus problem. Consensus is a fundamental problem in fault-tolerant distributed computing. Informally, n processes, each starting with a private value, are required to agree on one value chosen among their initial values. For shared memory systems, it is well known that *asynchronous fault tolerant* consensus is impossible as soon as at least one process may fail by *crashing* [28]. Trivially, consensus is thus impossible in anonymous, asynchronous and failure-prone shared memory. The same impossibility holds for non-anonymous message passing asynchronous systems [20].

Since the publication of this result, several approaches have been identified to overcome this impossibility, including randomization (e.g., [6]), strengthening the model with timing assumptions (e.g., [18]) or failure detectors (e.g., [12]) and strong synchronization primitives [25]. Similarly, in anonymous systems, randomization [10], failure detectors [7,14], as well as additional synchrony assumptions [16] have been investigated to solve consensus.

A *failure detector* is a distributed device which provides processes with possibly unreliable information about failures. Unreliable failure detectors, and more generally system assumptions which are not guaranteed to always hold, have motivated the study of *indulgent* algorithms [23]. Informally, an algorithm is indulgent if it is always *safe*, i.e., it never violates the safety part of the problem it is supposed to solve, and *converges to a decision* when the failure detector matches its eventual property. In this line of research, the key question is determining how fast indulgent algorithms converge when the eventual property of the failure detector is satisfied [17].

Contributions of the paper. This paper investigates the consensus problems in an anonymous, crash prone and asynchronous shared memory systems. In particular, we are interested in the individual *write* step complexity of anonymous consensus. Typically, shared memory systems use caching techniques to improve performances. When a write is performed, the system has to ensure that every cached copy is updated, which is costly. Differently, repeatedly reading a shared location may be a local operation. The paper presents the following two main results:

- The first result is a consensus algorithm. The set of input values that processes might propose is unbounded. The algorithm relies on a failure detector of the class $A\Omega$ [8] and tolerates up to $n-1$ process crashes. The "anonymous leader" class $A\Omega$ is the anonymous counterpart of the class Ω, which is the weakest failure detector for solving consensus [11] in the eponymous settings. Informally, when queried, a failure detector of the class $A\Omega$ returns

a boolean. Eventually, each query, except the queries issued by some non-faulty process, returns false. If no failure detector is available, we note that our algorithm can easily be made obstruction-free [26] by simply removing failure detector invocations. The algorithm is write-efficient in the following sense : a process executing solo decides after performing $O(\sqrt{n})$ write operations and $O(n)$ shared memory operations in total.

– The second result is a generalization of our consensus algorithm to the case of *homonymous* systems recently introduced by Delporte-Gallet et al [15], in which a small number c, $1 \leq c \leq n$ of identities is available. The system is no longer totally anonymous since processes have identities. However, when the number of ids is smaller than n, several processes may share the same id. The generalized algorithm achieves $O(\sqrt{n-c+1} + \log c/\log\log c)$ individual write complexity and $O(n - c + \log c/\log\log c)$ step complexity in solo execution. As in the case of anonymous systems, the algorithm relies on a failure detector of the class $A\Omega$ and the set of values that can be proposed is unbounded.

Roadmap. The paper is composed of 6 sections. Section 2 describes the anonymous shared memory model and the failure detector class $A\Omega$. An anonymous consensus is presented in Section 3. Its generalization to the case of systems with homonym processes follows (Section 4). Section 5 surveys related work and Section 6 concludes the paper.

2 System Model

Anonymous shared memory model. We consider a system Π of $n \geq 2$ deterministic processes. Processes are anonymous: they do not have identifiers, and they execute identical algorithms. The total number of processes n is however known by the processes. The system is asynchronous, in the sense that each process runs at its own speed, independently of the other processes.

Processes communicate with each other by reading and writing *atomic* shared registers (they are linearizable [27]). Registers are multi-writer and multi-reader: every register can be written in, or read from, by every process. In the pseudo-code we use to describe our algorithm, shared objects are denoted by upper-case letters, while lower-case identifiers are reserved for processes' local variables.

Failures and failure detectors. Processes may *crash*. A process is *correct* in an execution if it never crashes in this execution; otherwise it is *faulty*. We make no assumption on the number of crashes that may occur during a run.

As noted in the Introduction, a *failure detector* is a distributed oracle that provides processes with possibly unreliable information about failures [12]. Several classes of failure detectors suited to anonymous systems have been defined [8]. The failure detector we consider is anonymous Ω, denoted hereafter $A\Omega$. Each process is provided with a primitive $A\Omega.query()$, which returns *true* or *false*.

The following property, termed *eventual leadership* is ensured: there exists some correct process p_0 such that eventually every $A\Omega.query()$ always returns *true* at p_0, and *false* at every other process.

Consensus. Consensus is a distributed task which consists in a single operation $propose(v)$ that takes as input a value v in some (possibly unbounded) set $\mathbb{V}$, and returns a value v' in $\mathbb{V}$. When a process p invokes $propose(v)$, we say that p *proposes* v. Similarly, when $propose(v)$ returns a value v, we say that p *decides* v. Consensus requires that in every run: (Agreement) two processes cannot decide different values; (Validity) if a process decides some value v, then v was proposed before; and (Termination) every correct process eventually decides.

Time complexity. Consider an algorithm $\mathcal{A}$ that solves consensus in an asynchronous system equipped with an eventual failure detector such as $A\Omega$. In every execution, a correct leader process eventually emerges, but there is no bound on the time at which a correct process is elected. Obviously, the worst-case number of reads, or writes, performed by a process is unbounded. Thus, we measure the time complexity of asynchronous consensus algorithms in solo executions. Specifically, the *individual write complexity* (respectively the *individual step complexity*) is the worst-case number of write operations (respectively the total number of read aand write operation) that occur in solo executions in which only one process participates and this process is the leader output by the failure detector from the beginning of the execution.

3 The Janus[1] Algorithm

3.1 Description of Janus

The Janus algorithm solves consensus among n asynchronous and anonymous processes. Its pseudo-code is depicted in Figure 1. Janus relies on a failure detector of the class $A\Omega$ and tolerates up to $n-1$ process failures. No knowledge of the set of values that can be proposed is required. In particular, this set might be unbounded.

A process p initiates its algorithm by invoking $\mathsf{propose}(v)$, where v is the input value of p. Process p then launches two tasks **T1** and **T2** that run in parallel (line 1). In task **T2**, p monitors a shared register *decision* D, which is initialized to $\perp$[2]. If p reads a non-$\perp$ value d in D, p decides that value (line 21) and terminates.

In task **T1**, the execution proceeds in asynchronous rounds. Process p maintains an *estimate* (stored in the local variable est), which is the value it currently

[1] In Roman religion and mythology, Janus is the god of gates. Most often he is depicted as having two heads, facing opposite directions (Wikipedia). The choice of the name is explained by the fact that each process in our algorithm has to look in two directions: forward to check if another process has already started a new round, and back to check if another process concurrently executed the $\mathcal{K}$ past rounds.

[2] $\perp$ is a special value that is never proposed by the processes.

```
shared variables
        ∀r > 0 : T[r] is a multivalued MWMR atomic register, initially ⊥
        ∀r > 0 : C[r] is a binary MWMR atomic register, initially false
        D is a multivalued MWMR atomic register, initially ⊥

propose(v)
 (1)    est ← v; rnd ← 0; start T1; start T2;

task T1 :
 (2)    while (true) do
 (3)      if (AΩ-QUERY()) then
 (4)          rnd ← rnd + 1
              % Look for an estimate with higher priority %
 (5)          if (T[rnd] ≠ ⊥) then let r ← min{r' > rnd | T[r'] = ⊥} ;
 (6)                               est ← T[r − 1]; rnd ← r − 1
 (7)                          else T[rnd] ← est
 (8)          end if
              % Look for conflicting estimates in the last K rounds %
 (9)          for each i : 0 ≤ i < min(rnd, K) do
 (10)             if (T[rnd − i] ≠ est) then C[rnd − i] ← true end if
 (11)         end for
              % Check if no conflict occurs in the last K rounds %
 (12)         can_decide ← true;
 (13)         if (rnd ≥ K) then
 (14)             for each i : 0 ≤ i < K do
 (15)                 if (C[rnd − i] = true) ∨ (T[rnd − i] ≠ est) then can_decide ← false endif
 (16)             end for end if
 (17)         if (can_decide) then D ← est endif
 (18)     end if
 (19)   end while

task T2 :
 (20)   repeat d ← D until d ≠ ⊥
 (21)   stop T1; decide(d)
```

Fig. 1. The Janus algorithm, $\mathcal{K} = 2\left\lceil \sqrt{n} \right\rceil + 1$

favors. During each round to which it participates, p tries to *commit* its estimate by writing it in the decision register D (line 17). The algorithm ensures that (1) no two distinct values are committed and (2) at least one process eventually commits its estimate. To that end, each round r is associated with two multi-writer/multi-reader shared registers: the *value* register $T[r]$ and the *conflict* register $C[r]$. Intuitively, $T[r]$ stores a value that some process is willing to commit in round r, while $C[r]$, when set to *true*, indicates that two or more processes try to commit distinct values in round r.

A process p entering round r first checks whether a value has already been written in $T[r]$ (line 5). If this happens, p immediately enters round $r' \geq r$, where r' is the greatest round for which a value has been written to the associated register $T[r']$, thus possibly skipping rounds $r, \ldots, r' - 1$. In addition, p adopts

the value currently stored in $T[r']$ as its new estimate. Otherwise, i.e., when $T[r]$ equals $\perp$, p writes its estimate in $T[r]$.

Writing/reading value v to/from the value register $T[r]$ is however not sufficient to allow this value to be committed. Several processes may be performing write operations concurrently on $T[r]$ and thus, assuming that v is committed, a process entering round r later might adopt a value $v' \neq v$ and commits this value. Therefore, before committing its estimate v (that is, writing v in D, line 17), process p first checks that no conflicts have been detected in the last $\mathcal{K}$ rounds and that the registers $T[r], T[r-1], \ldots, T[r-\mathcal{K}+1]$ still store v (lines 14–16). For large enough values of $\mathcal{K}$, these two conditions prevent any other value different from v from being written in $T[r]$. We show in the proof (Lemma 6) that for $\mathcal{K} \geq \lceil 2\sqrt{n} \rceil + 1$ this property is ensured.

Conflicts are detected at lines 9–11. A process p with estimate v executing round r performs a read operation in every register $T[r'], r - \mathcal{K} + 1 \leq r' \leq r$. Whenever a value different from v is returned, the corresponding conflict register $C[r']$ is updated to *true*.

Finally, the progress of Janus relies on the underlying failure detector $A\Omega$. A process is allowed to enter round r only if it considers itself as a leader. In more details, before entering round r, each process queries its local failure detector module (line 3). Only if this query returns *true*, the process starts round r. Eventually, a unique non-faulty process is elected by the failure detector. This process eventually executes rounds alone, and eventually decides (See Lemma 3). When a failure detector is not available, we note that Janus is easily made obstruction-free by removing the query to the failure detector at line 3.

3.2 Proof of the Janus Algorithm

Fix some execution of the algorithm. Since the shared objects (i.e. the registers) are atomic the execution (as an interleaved sequence of reads and writes operation of the processes) is linearizable [27]. As a consequence, we may consider σ a linearization of the reads and writes operations. We shall say that an operation in σ on some register *occurs at time* τ if τ is the linearization point of that operation. As usual, we shall note var_p the local variable var of process p. The *execution of the (asynchronous) round* r *by* p is the interval during which $rnd_p = r$. More precisely, it is the sequence of steps applied by p when $rnd_p = r$. Missing proofs of Lemmata 1 and 1 can be found in the full version [9].

A process, executing round r, writes its estimate v in $T[r]$, provided it observes that no value has been previously written in $T[r]$ (line 7). The following Lemma implies that if this occurs, v has been previously written to $T[1], \ldots, T[r-1]$.

Lemma 1. *Let $r > 1$. Suppose that a write operation op with parameter v is performed on $T[r]$. Then a write operation op$'$ of value v to $T[r-1]$ occurs before op.*

It then follows from the previous Lemma that algorithm 1 satisfies the validity requirement of consensus.

Lemma 2 (Validity). *Every decided value is a proposed value.*

Termination then followed from the eventual leadership property of the failure detector $A\Omega$.

Lemma 3 (Termination). *Every correct process eventually decides.*

Proof. Assume for contradiction that some correct process q never decides. As, (1) only non-$\perp$ values can be written in D, and (2) q reads D infinitely many times and never decides, no value $v \neq \perp$ is written in D. As a process may decide only if it reads a value different from $\perp$ in D, this implies that no process decides.

By the eventual leadership property of the failure detector class $A\Omega$, there is a correct process p and a time τ such that each $A\Omega$-QUERY() performed after τ returns *true* if and only if the invoking process is p. At time τ, let R be the largest round such that $T[R-1] = \perp$. Clearly, p is the only process that can execute rounds $R+1, R+2, \ldots$ (line 3). Moreover by Lemma 1, for all $i > 0$, we have that $T[R+i] = \perp$.

As p is correct, it never decides, and for all $i > 0$ we have that $T[R+i] = \perp$, p eventually executes rounds $R+1, R+2, \ldots$ As p is the only process that executes those rounds, it follows from the code (lines 5–7) that p writes in each register $T[R+i]$ for all $i > 0$. Besides, it is not difficult to observe that the same value, say v, is written by p in each register $T[R+i]$.

As no process except p executes rounds $R+i, i > 0$, no process except p performs write operations on registers $T[R+i], i > 0$. Therefore it holds forever that $C[R+i] = \mathit{false}$ and $T[R+i] = v$, once p has written v in $T[R+i]$. Consider the execution of round $R+\mathcal{K}$ by p. Process p first writes v in $T[R+\mathcal{K}]$ (line 7). After this occurs, we have $C[R+i] = \mathit{false}$ and $T[R+i] = v$ for each $i, 0 < i \leq \mathcal{K}$. Hence, $\mathit{can\_decide}_p = \mathit{true}$ after the execution of the **for each** loop at lines 14–16. We conclude that p writes v in D (line 17), and decides by the code of task **T2**: contradiction. □

Proof of agreement. We divide the execution in *epochs* as follows. Epoch e_i is an interval that starts with the first write (according to the linearization σ) to register $T[i]$ and ends immediately before the first write (if any) performed to register $T[i+1]$. Given a read, or write, operation *op*, we say that *op occurs in epoch* e_i, or equivalently, that *op is performed in* e_i, if *op* is linearized in the interval e_i. Clearly, if a write to $T[j]$ occurs in e_i, then $j \leq i$. The next lemma directly follows from the code of Janus (lines 5 and 7).

Lemma 4. *Suppose that p performs a write operation op on $T[i]$. The last operation preceding op performed by p is a read on $T[i]$, and the value returned by that operation is $\perp$.*

Suppose that process p performs a write operation on register $T[j]$ in epoch e_i. When this operation terminates, a value has already been written in $T[i]$ by definition of e_i. Lemma 4 then implies that the next write operation by p (if any) is performed on some register $T[j']$ such that $j' > i$. Lemma 5 bellow captures precisely this observation.

Lemma 5. *Denote by op, op' two write operations performed by the same process p. Suppose that: (1) op occurs in e_i, (2) op' is a write on register $T[j]$ with $j \neq i$, and (3) op precedes op'. Then, $j > i$.*

Proof. By Lemma 4, p reads from $T[j]$ immediately before executing op', and this read operation returns $\perp$. Let op'' denote that operation. It follows from the third condition of the Lemma that op'' occurs after op, which in turn occurs after some non-$\perp$ value has been written in $T[i']$ for each $i' \leq i$ (By definition of e_i, and the fact op occurs in e_i.). Since the read operation op'' performed on $T[j]$ returns $\perp$, we conclude that $j > i$. □

Consider a round number r, and a value v. We say that value v is *committed at round r* if there exists a process p that writes v in D (line 17) while it is executing round r. Observe that in such a case, v is the estimate of p, and v has been written in $T[r]$ (by p itself or some other process). Note moreover that for each decided value v, there exists a round during which v is committed.

The following lemma is central to the proof of the agreement property. Informally, this lemma says that if some process writes a value v in the decision register D while executing round r, no other value than v can be written to $T[r]$.

Lemma 6. *Let v be a value, and R be a round number such that v is committed at round R. For every value v' written in $T[R]$, it holds that $v' = v$.*

The agreement property then follows by combining Lemma 1 and Lemma 6, and observing that every decided value has been committed.

Lemma 7 (Agreement). *No two process decide different values.*

Proof. Let v and v' be two decided values (at line 21). By the code of Algorithm 1, v and v' have been previously written in D (at line 17). Hence, v and v' are committed at some round, say, r and r' respectively. Without loss of generality, assume that $r \leq r'$. Let p' be a process that writes v' in D in round r'. Observe that v' is the estimate of p' in round r'. Therefore, v' has been written in $T[r']$, either by p' (at line 7) or by some other process (in the latter case, v' was read by p' at line 6). As $r \leq r'$, it follows from Lemma 1 that v' is written in $T[r]$ as well. Since v is committed at round r, we conclude by Lemma 6 that $v = v'$. □

The rest of this section is devoted to the proof of Lemma 6. We proceed by contradiction. We name H the following assumption:

> There exists a round R such that two write operations with parameters $u \neq v$ are performed on $T[R + \mathcal{K}]$ and v is committed in round $R + \mathcal{K}$.

In the following, we show that to satisfy assumption H the system must consist of at least $n + 1$ processes.

Denote by R the round number appearing in assumption H. For each $i, j, 1 \leq i, j \leq \mathcal{K}$, note W_j^i the set of processes that perform a write operation to register $T[R+j]$ during epoch e_i. More precisely, a process p belongs to W_j^i if and only if there exists a write operation to $T[R+j]$ by p which occurs in e_i. By the definition of epochs, we know that if $j > i$, then $W_j^i = \emptyset$. The three lemmata below further precise how the sizes of the W_j^i's and the round numbers are related.

Lemma 8. *If assumption H holds, then:* $\forall i, 1 \leq i < \mathcal{K},\ |W_i^i| \geq 2$.

Proof. By assumption H, at least two values v and u are written in $T[R+\mathcal{K}]$. It follows from Lemma 1 that v and u must have been written in $T[R+i]$ for each i such that $1 \leq i < \mathcal{K}$. It remains to show that such a write operation with parameter v (resp. u) occurs in e_i.

Let us consider the first write of v in $T[R+i]$. Clearly, this operation occurs in epoch $e_{R+i'}$, for some $i' \geq i$. Suppose for the sake of contradiction that $i' > i$. Hence, the first time v is written in $T[R+i]$, a value has already been written in $T[R+i+1]$. Let p be the process that performs this first write of v in $T[R+i+1]$. As v is written to $T[R+\mathcal{K}]$, p must exist by Lemma 1. Denote $w_p(R+i+1)$ the write operation of p. According to the code of Janus we know that: (1) p performs that operation while it is executing round $R+i+1$ (line 7), (2) $w_p(R+i+1)$ is preceded by a read operation of $T[R+i+1]$ (denoted $r_p(R+i+1)$) by p that returns $\perp$, and (3) in round $R+i$, there is a read operation from $T[R+i]$ that returns v or a write of v by p to $T[R+i]$. Denote by $op_p(R+i)$ this last operation, and $op_p(R+i), r_p(R+i+1), w_p(R+i+1)$ the operations that occur in this order. Moreover, $op_p(R+i)$, which reads or writes v in $T[R+i]$ occurs in epoch $e_{R+i''}$ for some $i'' \geq i'$, since the write of v in $T[R+i]$ occurs in $e_{R+i'}$. Therefore, operation $r_p(R+i+1)$ occurs after a write in $T[R+i+1]$, from which we conclude that $r_p(R+i+1)$ returns a non-$\perp$ value. It thus follows by Lemma 4 that p does not write in $T[R+i+1]$: a contradiction.

We have shown that a write of v in $T[R+i]$ occurs in epoch e_i. A similar argument applied to value u yields that a write of u in $T[R+i]$ occurs in e_i. Since each process does not write twice in the same register, $|W_i^i| \geq 2$. □

Lemma 9. *If assumption H holds, then :* $\forall i, j : 1 \leq i < \mathcal{K}$ *and* $1 \leq j < i$, $|W_j^i| \geq 1$.

Proof. We start by establishing that two read operations that return v and u respectively occur in e_i.

As v is written in $T[R+\mathcal{K}]$, v is also written in $T[R+i+1]$ (Lemma 1). Let p the process that performs the first write of v in $T[R+i+1]$. By the code, p executes round $R+i$ before performing that write operation, and v is the estimate of p in that round. At the beginning of round $R+i$, p either reads v in $T[R+i]$ or writes v in $T[R+i]$. Moreover, the read operation on $T[R+i+1]$ performed by p at the beginning of round $R+i+1$ returns $\perp$ (Otherwise p does not perform a write operation on $T[R+i+1]$). Therefore, every operation performed by p while it is executing round $R+i$ occurs in epoch e_{R+i}.

In particular, the read of $T[R+j]$ performed by p at line 10 occurs in e_{R+i}. This read must return v. Otherwise, p writes *true* in $C[R+i]$, and this operation occurs in e_{R+i}. As no process ever writes *false* in $C[R+i]$, every read operation performed on $C[R+i]$ that occurs in later epochs return *true*. Consider a process p' executing round $R+\mathcal{K}$. p' reads $C[R+i]$ at line 15. This read operation occurs after a write operation has been performed on $T[R+\mathcal{K}]$, so it occurs after the end of epoch e_{R+i}. Hence, that operation returns *true* and thus p' cannot write in D in that round. Therefore, no value is committed in round $R+\mathcal{K}$, contradicting assumption H.

Similarly, by considering the process that performs the first write of u in $T[R+i+1]$, we get that a read operation of $T[R+j]$ that returns u occurs in e_{R+i}.

Finally, as there are two read operations of $T[R+j]$ returning two different values occur in e_i, there must exist a write operation on $T[R+j]$ that occurs in e_i. We thus conclude that $W_j^i \neq \emptyset$. □

Lemma 10. *Suppose that assumption H holds. Let* i, i', j, j' *such that* $1 \leq i \leq i' < \mathcal{K}$ *and* $1 \leq j < i$, $1 \leq j' < i'$. $W_j^i \cap W_{j'}^{i'} \neq \emptyset \Rightarrow (i = i' \wedge j = j') \vee (i < j')$

Proof. Let $p \in W_j^i \cap W_{j'}^{i'}$. By definition, a write operation by p occurs in e_i and $e_{i'}$. Either $i = i'$ and $j = j'$ or, by Lemma 5, $i < j'$. □

Proof of Lemma 6. Assume for the sake of contradiction that assumption H is satisfied, and consider the following set:

$$S = \left\{ (i,j) : \left\lceil \frac{\mathcal{K}-1}{2} \right\rceil \leq i \leq \mathcal{K}-1, 1 \leq j \leq \left\lceil \frac{\mathcal{K}-1}{2} \right\rceil \right\}$$

In what follows, we count the total number of processes that appear in the union of the sets W_j^i, where $(i,j) \in S$, then we show that this union includes at least $n+1$ distinct processes.

Let $(i,j) \neq (i',j') \in S$ such that $i \leq i'$. By definition of S, $i \geq j'$ and thus it follows from Lemma 10 that $W_j^i \cap W_{j'}^{i'} = \emptyset$. Hence,

$$\left| \bigcup_{(i,j)\in S} W_j^i \right| = \sum_{(i,j)\in S} |W_j^i|$$

Moreover, It follows from Lemmas 8 and 9 that $|W_j^i| \geq 1$ for each $(i,j) \in S$ and $|W_i^i| \geq 2$ for each $(i,i) \in S$. Therefore,

$$\left| \bigcup_{(i,j)\in S} W_j^i \right| \geq \left\lceil \frac{\mathcal{K}-1}{2} \right\rceil \cdot \left\lceil \frac{\mathcal{K}-1}{2} \right\rceil + 1$$

Finally, as $\mathcal{K} = 2 \cdot \lceil \sqrt{n} \rceil + 1$, we get $\left| \bigcup_{(i,j)\in S} W_j^i \right| \geq n+1$. Therefore, assuming that H is satisfied, we have exhibited a set of $n+1$ distinct processes : a contradiction. Consequently, H cannot be satisfied, from which we conclude that no value different from v is written in $T[R]$, as desired. □

Theorem 1. *The Janus algorithm described in Figure 1, when instantiated with a failure detector of the class $A\Omega$ solves consensus in an n-processes, anonymous shared memory system.*

Proof. Immediately follows from Lemmas 2, 3 and 7. □

The following theorem proves that the step complexity of Janus is $O(n)$, which is optimal [4], and that its write complexity equals to $O(\sqrt{n})$.

Theorem 2. *The Janus algorithm has a step complexity of $O(n)$, and a write complexity of $O(\sqrt{n})$.*

Proof. Consider a solo execution of some process p. During this execution, p executes $\mathcal{K} = 2\lceil\sqrt{n}\rceil + 1$ rounds, then decides. Name $\{1, \ldots, \mathcal{K}\}$ the rounds executed by p, and consider some round i. According to the code of Algorithm 1, during round i process p executes a single write (line 7), and reads $3i + 1$ shared registers (lines 5, 9 to 11, and 14 to 16). As a consequence, the step complexity of Janus is $O(n)$, and its write complexity equals $O(\sqrt{n})$. □

4 The Case of Homonymous Systems

In an homonymous system, $c, 1 \leq c \leq n$ identities are available [15,30]. Each process has an identifier in the range $\{1, \ldots, c\}$. Processes that share the same identifier are said to be homonym, and for each $i \in \{1, \ldots, c\}$, there is at least one process with id i (and thus at most $n - c + 1$).

In this section we present a consensus algorithm for homonymous shared-memory systems that tolerates up to $n - 1$ process failures. As in the case of anonymous systems, the algorithm relies on a failure detector of the class $A\Omega$ and the set of values that can be proposed is unbounded. The algorithm is built in a modular way from several copies of the Janus algorithm and an efficient implementation of m-valued adopt-commit objects due to Aspnes and Ellen [4].

Adopt-commit. An adopt-commit object [21] is a shared object that supports a single operation denoted $propose(v)$ where v is a value taken from some set $\mathbb{V}$. Every invocation of $propose(\cdot)$ returns a response of the form (b, v') where $b \in \{commit, adopt\}$ and $v' \in \mathbb{V}$ such that the following properties hold: (Termination) Every invocation of $propose(\cdot)$ by a correct process terminates; (Validity) If (b, v) is returned, then some process previously invoked $propose(v)$; (Agreement) If $(commit, v)$ is returned, then every decision has the form $(*, v)$; (Convergence) If every process proposes the same value v, then $(commit, v)$ is the only possible decision.

An efficient crash-tolerant asynchronous implementation of m-valued adopt-commit objects from multi-reader multi-writer registers in anonymous system is presented by Aspnes and Ellen in [4]. The algorithm achieves $O(\frac{\log m}{\log \log m})$ individual step-complexity provided that the set $\mathbb{V}$ from which proposed values are taken is a priori known contains at most m values.

Overview of the algorithm. The algorithm, described in Figure 2, proceeds in asynchronous rounds. Each round is divided in two phases, an *agreement* phase in which each group of homonym processes agree on a common value, and a *conciliation* phase in which processes check whether every group agrees on the same value.

The agreement phase of round r is implemented by c instances of the Janus algorithm that we note $J[r][1], \ldots, J[r][c]$. As in the Janus algorithm, each process maintains an estimate stored in the local variable est. Processes with identity id propose their estimate to the same instance of Janus $J[r][id]$ (line 4). The array $V[r][1..c]$ is then used to store the decisions that occur (if any) in each of the c instances $J[r][1..c]$ (line 5). This completes the agreement phase of round r.

Note that each instance of Janus is implemented with its own collection of registers. Processes however share a single failure detector $A\Omega$. This means that a given instance of Janus might not progress if no process participating in this instance is elected by the failure detector. Nevertheless, if every correct process participates in at least one of the Janus instances of round r, termination is ensured in at least one instance, namely the instance $J[r][id]$, where id is the identity of the eventual leader. The conciliation phase of round r is implemented by a single adopt-commit object denoted $AC[r]$. A process p with identity id that has previously obtained a decision d from the instance of Janus $J[r][id]$

```
shared objects
        ∀r > 0 : J[r][1..c] is an array of c copies of Janus
        ∀r > 0 : AC[r] is an adopt-commit object
        ∀r > 0 : V[r][1..c] is an array of c MRMW registers, initially ⊥
        DD is a multivalued MWMR atomic register, initially ⊥

propose(v)
 (1)    est ← v; rnd ← 0; start T1; start T2;

task T1 :
 (2)    while (true) do
 (3)       rnd ← rnd + 1;
 (4)       est ← J[rnd][id].propose(est);
 (5)       V[rnd][id] ← est;
 (6)       (b, id′) ← AC[rnd].propose(id);
 (7)       est ← V[rnd][id′];
 (8)       if b = commit then DD ← est endif
 (9)    end while

task T2 :
 (10)   repeat d ← DD until d ≠ ⊥
 (11)   stop T1; decide(d)
```

Fig. 2. Consensus with homonyms, code for processes with identity id

and has written this value to the register $V[r][id]$ checks whether it is safe to decide this value. To do so, it proposes its identity to the adopt-commit object $AC[r]$ (line 6). Let (b, id') denote the response of the object obtained by p. p first adopts the value it reads from $V[r][id']$ as its new estimate (line 7). Note that the read operation of $V[r][id']$ returns a non-$\perp$ value. This is because by the validity property of adopt-commit, a process p' with identity id' must have proposed its identity to $AC[r]$ before p obtains the response (b, id'). In addition, before accessing $AC[r]$, p' must have written a value to $V[r][id']$.

Second, if $b = commit$, p then writes its estimate in the shared register DD, indicating that this value can be safely decided. Indeed, by the agreement property of adopt-commit, every propose() operation to $AC[r]$ returns $(adopt, id')$ or $(commit, id')$. Hence, as a unique value v is written in $V[r][id']$, the estimate of each process that completes round r is equal to v. It thus follows that v is the only value that may be written to DD in round r and any subsequent round.

Termination relies on the underlying failure detector $A\Omega$. The eventual leadership property ensures that after some time τ, a single correct process considers itself as a leader. Let id denote the identity of this eventual leader. Observe that, by the code of Janus, a process participating in the execution of an instance of Janus does not take write steps unless it considers itself as a leader (Figure 1, line 3). Therefore, no decisions occur in every instance $J[r][id']$ that starts after τ if $id' \neq id$. On the other hand, every instance $J[r'][id]$, $r' \geq 1$ eventually produces a decision because the set of processes that participate in these instances includes the eventual leader. Consequently, if each round r instance of Janus starts after τ, only process with identity id may access the object $AC[r]$. Since they all propose the same value, namely id, it follows from the convergence property of adopt-commit that they get back $(adopt, id)$. This implies that a value is eventually written to the decision register DD, and termination follows.

Complexity. Since at most $n-c+1$ processes participate in each instance of Janus ($n-c+1$ is the maximal size of a group of homonym processes), the parameter $\mathcal{K}$ is set to $2\sqrt{n-c+1}+1$ in each instance. Values proposed to objects $AC[r]$ are always taken from the set of available identities $\{1, \ldots, c\}$. Each adopt-commit object is thus implemented by the optimal algorithm by Aspnes and Ellen [4]. A process executing solo, and elected leader by the failure detector from the beginning of the execution, decides after participating in one instance of Janus, and performing one propose() operation on an adopt-commit object. In addition, it performs two write operations (at lines 5 and 8). Therefore, in solo executions, the individual write complexity equals to $O(\sqrt{n-c+1} + \frac{\log c}{\log \log c})$ and the individual step complexity equals to $O(n-c+1+\frac{\log c}{\log \log c})$.

Proof. The correctness proof of the algorithm described in Figure 2 is presented in a companion technical report [9].

5 Related Work

Attiya et al. [5] characterized *failure-free* tasks that are solvable using registers when the number of processes n is unknown. In particular, the authors show, using bivalence and covering arguments, that consensus in such an environment requires more than $\Omega(\log n)$ atomic registers, and at least $\Omega(\log n)$ total work. Recently, Aspnes and Ellen [4] proved that the individual step complexity of adopt-commit object in anonymous shared-memory is $\Theta(min\ (\frac{\log m}{\log \log m}, n))$, where m is the number of different values that might be proposed to the object. Because consensus satisfies the specification of an adopt-commit object [21], this lower bound also holds for the consensus object.

Guerraoui and Ruppert [24] studied the computational power of shared memory distributed systems in the presence of both anonymity and failures. They propose constructions for several fundamental abstractions: wait-free timestamping and snapshots, and obstruction-free consensus. In particular, the authors depict an anonymous *binary* consensus algorithm having a step complexity of $O(1)$. *When m is known*, this algorithm solves anonymous consensus in $O(\log m)$ write operations and $O(\log m)$ individual work. Delporte-Gallet and Fauconnier [14] proposed an anonymous consensus which relies on failure detector $A\Omega$ and a weak set abstraction. If m is known, this algorithm solves consensus in $O(\log m)$ individual work and $O(1)$ writes.

Abrahamson [1] studied binary consensus in the probabilistic-write model with eponymous processes, when identities are only used to label registers. Recently, Aspnes [3] proposed a consensus algorithm for the probabilistic-write anonymous model which solves consensus in $O(\log m)$ individual work. The algorithm is based on the decomposition of consensus into two distinct components: an adopt-commit object which detects agreement, and a conciliator, which ensure agreement with some probability. Aside from their lower bound result, the authors of [4] proposed two asymptotically optimal implementations of adopt-commit objects: a $O(\frac{\log m}{\log \log m})$ solution which requires that m is known, and a $O(n)$ solution which solves the problem without any assumptions over m. During a solo execution, the latter algorithm writes in $O(n)$ different registers.

The Janus algorithm we depicted in Section 3 solves anonymous consensus in $O(n)$ individual work, and $O(\sqrt{n})$ write operations, a result which matches the lower bound of [3] and further improves the write complexity of anonymous consensus.

The notion of partial anonymity in which some processes may share the same identifier was first introduced by Yamashita et al. [30] in the context of the leader election problem. The term homonyms was coined recently by Delporte et al. [15]. In this work, the authors study the Byzantine consensus problem in message passing systems when a limited number of identities is available.

6 Conclusion

This paper has presented two efficient consensus algorithms for anonymous and partially anonymous asynchronous shared memory systems. Both algorithms do

not impose restrictions on the set $\mathbb{V}$ from which proposed values are taken. The complexity depends solely on the number of processes n and the number of available identifiers c in the partially anonymous case. To the best of our knowledge, the generalized algorithm presented in Section 4 is the first non-trivial consensus implementation for shared memory homonymous systems.

Of note, by limiting the Janus algorithm to its first $\mathcal{K}$ rounds and removing the queries to the failure detector, we obtain an anonymous adopt-commit implementation whose individual write complexity is $O(\sqrt{n})$, while retaining an optimal $O(n)$ individual work. With respect to the write complexity, this is an improvement over existing implementations.

This paper focuses on consensus algorithms for which the set of input values is not restricted. A direction for future research is to investigate the interplay between the size of the input set m, the number of available identifiers c, the number of processes n, and the number of distinct values k that can be decided.

References

1. Abrahamson, K.: On achieving consensus using a shared memory. In: Proc. of the 17th Symp. on Principles of Distributed Computing (PODC), pp. 291–302. ACM (1988)
2. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. Distributed Computing 18(4), 235–253 (2006)
3. Aspnes, J.: A modular approach to shared-memory consensus, with applications to the probabilistic-write model. In: Proc. of the 29th Symp. on Principles of Distributed Computing (PODC), pp. 460–467. ACM (2010)
4. Aspnes, J., Ellen, F.: Tight bounds for anonymous adopt-commit objects. In: Proc. of the 23rd Symp. on Parallelism in Algorithms and Architectures (SPAA), pp. 317–324. ACM (2011)
5. Attiya, H., Gorbach, A., Moran, S.: Computing in totally anonymous asynchronous shared memory systems. Inf. Comput. 173, 162–183 (2002)
6. Ben-Or, M.: Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In: Proc. of the 2nd Symp. on Principles of Distributed Computing (PODC), pp. 27–30. ACM (1983)
7. Bonnet, F., Raynal, M.: The Price of Anonymity: Optimal Consensus Despite Asynchrony, Crash and Anonymity. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 341–355. Springer, Heidelberg (2009)
8. Bonnet, F., Raynal, M.: Anonymous Asynchronous Systems: The Case of Failure Detectors. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 206–220. Springer, Heidelberg (2010)
9. Bouzid, Z., Sutra, P., Travers, C.: Anonymous Agreement: The Janus Algorithm. Technical report, `http://hal.inria.fr/inria-00625704/en/`
10. Buhrman, H., Panconesi, A., Silvestri, R., Vitányi, P.M.B.: On the importance of having an identity or, is consensus really universal? Distributed Computing 18(3), 167–176 (2006)
11. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM 43(4), 685–722 (1996)

12. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM 43(2), 225–267 (1996)
13. Chothia, T., Chatzikokolakis, K.: A Survey of Anonymous Peer-to-Peer File-Sharing. In: Enokido, T., Yan, L., Xiao, B., Kim, D.Y., Dai, Y.-S., Yang, L.T. (eds.) EUC-WS 2005. LNCS, vol. 3823, pp. 744–755. Springer, Heidelberg (2005)
14. Delporte-Gallet, C., Fauconnier, H.: Two Consensus Algorithms with Atomic Registers and Failure Detector Ω. In: Garg, V., Wattenhofer, R., Kothapalli, K. (eds.) ICDCN 2009. LNCS, vol. 5408, pp. 251–262. Springer, Heidelberg (2008)
15. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Kermarrec, A.M., Ruppert, E., Tran-The, H.: Byzantine agreement with homonyms. In: Proc. of the 30th Symp. on Principles of Distributed Computing (PODC), pp. 21–30. ACM (2011)
16. Delporte-Gallet, C., Fauconnier, H., Tielmann, A.: Fault-tolerant consensus in unknown and anonymous networks. In: Proc. of the 29th Int'l Conference on Distributed Computing Systems (ICDCS), pp. 368–375. IEEE (2009)
17. Dutta, P., Guerraoui, R.: The inherent price of indulgence. Distributed Computing 18(1), 85–98 (2005)
18. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM 35(2), 288–323 (1988)
19. Federrath, H. (ed.): Designing Privacy Enhancing Technologies. LNCS, vol. 2009. Springer, Heidelberg (2001)
20. Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (1985)
21. Gafni, E.: Round-by-round fault detectors: unifying synchrony and asynchrony. In: Proc. of the 17th Symp. on Principles of Distributed Computing (PODC), pp. 143–152. ACM (1998)
22. Guerraoui, R., Ruppert, E.: What can be implemented anonymously? In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 244–259. Springer, Heidelberg (2005)
23. Guerraoui, R., Lynch, N.A.: A general characterization of indulgence. TAAS 3(4) (2008)
24. Guerraoui, R., Ruppert, E.: Anonymous and fault-tolerant shared-memory computing. Distributed Computing 20(3), 165–177 (2007)
25. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. 13(1), 124–149 (1991)
26. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: Proc. of the 23rd Int'l Conference on Distributed Computing Systems (ICDCS), pp. 522–529. IEEE (2003)
27. Herlihy, M., Wing, J.: Linearizability: a correcteness condition for concurrent objects. ACM Trans. on Prog. Lang. 12(3), 463–492 (1990)
28. Loui, M., Abu-Amara, H.: Memory requirements for agreement among unreliable asynchronous processes. Advances in Computing Research 4, 163–183 (1987)
29. Ruppert, E.: The Anonymous Consensus Hierarchy and Naming Problems. In: Tovar, E., Tsigas, P., Fouchal, H. (eds.) OPODIS 2007. LNCS, vol. 4878, pp. 386–400. Springer, Heidelberg (2007)
30. Yamashita, M., Kameda, T.: Leader election problem on networks in which processor identity numbers are not distinct. IEEE Transactions on Parallel and Distributed Systems 10(9), 878–887 (1999)