On the Optimal Space Complexity of Consensus for Anonymous Processes

Rati Gelashvili

On the Optimal Space Complexity of Consensus for Anonymous Processes

Rati Gelashvili

MIT
gelash@mit.edu

Abstract. The optimal space complexity of consensus in shared memory is a decades-old open problem. For a system of n processes, no algorithm is known that uses a sublinear number of registers. However, the best known lower bound due to Fich, Herlihy, and Shavit requires $\Omega(\sqrt{n})$ registers.

The special symmetric case of the problem where processes are anonymous (run the same algorithm) has also attracted attention. Even in this case, the best lower and upper bounds are still $\Omega(\sqrt{n})$ and $O(n)$. Moreover, Fich, Herlihy, and Shavit first proved their lower bound for anonymous processes, and then extended it to the general case. As such, resolving the anonymous case might be a significant step towards understanding and solving the general problem.

In this work, we show that in a system of anonymous processes, any consensus algorithm satisfying nondeterministic solo termination has to use $\Omega(n)$ read-write registers in some execution. This implies an $\Omega(n)$ lower bound on the space complexity of deterministic obstruction-free and randomized wait-free consensus, matching the upper bound and closing the symmetric case of the open problem.

1 Introduction

The celebrated Fischer, Lynch and Paterson (FLP) [FLP85] result proved that fundamental synchronization tasks including consensus and test-and-set are not solvable in a wait-free manner using read-write registers. However, the work of Ben-Or [BO83] shows that it is possible to circumvent FLP and obtain efficient distributed algorithms, if we relax the problem specification to allow probabilistic termination. It is also possible to solve these tasks deterministically, but obstruction-free instead of wait-free; it is known how to convert any deterministic obstruction-free algorithm into a randomized wait-free algorithm against an oblivious adversary (see [GHHW13]).

The space complexity of an algorithm is the maximum number of registers used in any execution. A lot of research has been dedicated to improving the upper and lower bounds on the space complexity for canonical tasks. For test-and-set, an $\Omega(\log n)$ lower bound was shown in [SP89] and independently in [GW12]. On the other hand, an $O(\sqrt{n})$ deterministic obstruction-free upper bound was

given in [GHHW13]. The final breakthrough was the recent obstruction-free algorithm designed by Giakkoupis et al. [GHHW14], with $O(\log n)$ space complexity, essentially closing the problem[1].

For consensus, an upper bound with n registers was long known from [AH90]. A lower bound of $\Omega(\sqrt{n})$ by Fich et al. [FHS98] first appeared in 1993. The proof is notorious for its technicality and utilizes a neat inductive combination of covering and valency arguments. Another version of the proof appeared in a textbook [AE14]. However, a linear lower bound or a sublinear space algorithm has remained elusive to date.

The authors of [FHS98] conjectured a tight lower bound of $\Omega(n)$. But the linear lower bound has not been proven even in a restricted, symmetric case, where all processes are anonymous. In such a system processes can be thought of as running the same code: all processes with the same input start in the same initial state and behave identically. The same linear upper bound holds for anonymous processes, since a deterministic obstruction free consensus algorithm that uses $O(n)$ registers is known [GR05]. Interestingly, the proof in [FHS98] starts by showing the $\Omega(\sqrt{n})$ lower bound for anonymous processes, which is then extended to a much more complex argument for the general case. Therefore, a linear lower bound in the anonymous setting might prove to be a meaningful step in better understanding and solving the general case of the open problem.

Contribution: In this paper we prove the $\Omega(n)$ lower bound in the symmetric (anonymous) case for consensus algorithms satisfying the standard *nondeterministic solo termination* property. Any lower bound for algorithms satisfying the nondeterministic solo termination implies a lower bound for deterministic obstruction-free and randomized wait-free algorithms. As in [FHS98, AE14], the bound is for the worst-case space complexity of the algorithm, i.e. for the number of registers used in some execution, regardless of its actual probability.

Our argument relies on a specific class of executions which we call *reserving*, and on the ability to define valency, corresponding to possible return values, for these executions. This definition of valency and the ability to cover registers with modified contents by reserved processes greatly simplifies the task of performing an inductive argument. We hope these techniques will be useful for future work.

We also show how the lower bound can be extended to a non-anonymous, adaptive, setting where processes come from a very large namespace and the bound depends on the size of the subset of processes that actually participate in the execution. However, this extension requires additional restrictions on register size and termination, and is provided mainly to illustrate an approach.

Definitions and Notation: We use the standard shared-memory model and similar notation to [FHS98, AE14]. We consider anonymous processes and atomic read-write registers. A process is *covering* a register R, if the next step of p can be a write to R. A *block write* of a set of processes P to a set of covered registers

[1] The space complexity of randomized test-and-set against a strong (adaptive) adversary remains open.

V is a sequence of write steps by processes in P, where each step is a write to a different register and all registers get written to.

In a system of anonymous processes, if a process p in state s performs a particular operation, for any configuration with any process q in the same state s, q can also perform the exact same operation. Finally, if p and q perform the same operation from the same state with the same outcome (i.e. read the same value), then both p and q end up in the same state after the operation. In randomized algorithms, anonymous processes always perform the same operation from the same state (including flipping coins with the same random distribution), and end up in identical state if they observe the same results.

A *clone* of a process p, exactly as in [FHS98, AE14], is defined as another process with the same input as p, that shadows p by performing the same operations as p in lockstep, reading and writing the same values immediately after p, and remaining in the same state, all the way until some write of p. Because the system consists of anonymous processes, in any execution with sufficiently many processes, for any write operation of p, there always exists an alternative execution with a clone q that shadowed p all the way until the write. In particular, in the alternative execution, process q *covers* the register and is about to write the value that p last wrote there. Moreover, the two executions with or without the clone covering the register are completely indistinguishable to all processes other than the clone itself.

An execution is a sequence of steps by processes and a *solo* execution is an execution where all steps are taken by a single process. An execution interval is a subsequence of consecutive steps from some execution. In the binary consensus problem each participating process starts with a binary input 0 or 1, and must return a binary output. The correctness criterium is that all outputs must be the same and equal to the input of some process. We say that an execution interval *decides* 0 (or 1) if some process returns 0 (or 1, respectively) during this execution interval.

A wait-free termination requirement means that each participating process must eventually return an output within a finite number of own steps, regardless of how the other processes are scheduled. The FLP result shows that in the asynchronous shared memory model with read-write registers, no deterministic algorithm can solve binary consensus in a wait-free way. However, it is possible to deterministically solve obstruction-free consensus, i.e. when processes are only required to return an output if they run solo from some configuration. It is also possible to solve consensus in a randomized wait-free way, when processes are allowed to flip random coins and decide their next steps accordingly. A *nondeterministic solo termination* property of an algorithm means that from each reachable configuration, for each process, there exists a finite solo execution by the process where it terminates and returns an output. We prove our lower bounds for binary consensus algorithms that satisfy this *nondeterministic solo termination* property, because both deterministic obstruction-free algorithms and randomized wait-free algorithms fall into this category.

2 Space Complexity Lower Bound

In order to demonstrate our approach, we start by presenting a different proof of the $\Omega(\sqrt{n})$ space lower bound in the anonymous setting. It uses induction on the number of registers written during an execution, as opposed to induction on the tuple of sizes of pending block writes in [FHS98]. The proof also has an additional benefit that the use of covering and valency arguments is decoupled. As usual, we use covering to enforce writing to a new register, while a valency argument reminiscent of [FLP85] ensures that both decision values remain reachable by solo executions.

Next, building upon this new argument, we prove an $\Omega(n)$ space lower bound for consensus with nondeterministic solo termination in a system of anonymous processes. There are some significant differences, for instance, the execution is constructed in such a way that after a register is written to, it always remains covered. Moreover, valency is redefined to account for this specific class of executions. The rest is induction.

2.1 A Square-Root Lower Bound

In this section, we define *valency* as follows. If there is a solo execution of some process returning 0 from a configuration, then we call this configuration *0-valent* (and *1-valent* if there is a solo execution of a process that returns 1). Solo termination implies that every configuration is 0-valent or 1-valent. Note that unlike the standard definition of valency, our definition allows the same configuration to be simultaneously 0-valent and 1-valent. We call such configurations that are both 0-valent and 1-valent *bivalent*, and *univalent* otherwise. Notice that a configuration is bivalent if two solo executions of the same process return different values. If a configuration is 0-valent, but not 1-valent (i.e. no solo execution from this configuration decides 1), then we call it 0-univalent, meaning that the configuration is univalent with valency 0. Analogously, a configuration is 1-univalent if it is 1-valent but not 0-valent.

Observe that if we have at least two processes, then in every bivalent configuration we can always find two distinct processes p and q, such that there is a solo execution of p returning 0 and a solo execution of q returning 1. This is because either the configuration is bivalent because of solo executions of distinct processes, in which case we are done, or two solo executions of some process return different values, in which case it suffices to consider any terminating solo execution of another process.

For the system of anonymous processes, and a consensus algorithm that uses atomic read-write registers and satisfies the nondeterministic solo termination property, we prove the following statement by induction:

Lemma 1. *For $r \geq 0$, there exists a system of $\frac{(r-1)r}{2} + 2$ anonymous processes, such that for any consensus algorithm, a configuration C_r is reachable by an execution E_r with the following properties:*

– *There is a set R of r registers, each of which has been written to during E_r, and*
– *the configuration C_r is bivalent.*

Proof. The proof is by induction, with the base case $r = 0$. Our system consists of two processes p and q, p starts with input 0, q starts with input 1, and C_0 is the initial state. Clearly, no registers have been written to in C_0 and bivalency follows by nondeterministic solo termination.

Now, let us assume the induction hypothesis for some r and prove it for $r+1$. By the induction hypothesis, we can reach a configuration C_r using $\frac{(r-1)r}{2} + 2$ processes. The goal is to use another r processes and extend C_r to C_{r+1}, completing the proof since $r + \frac{(r-1)r}{2} + 2 = \frac{r(r+1)}{2} + 2$.

As discussed above, because we have at least 2 processes and C_r is bivalent, there exists a process p and its solo execution α from C_r after which p returns 0 and a process $q \neq p$ and its solo execution β from C_r after which q returns 1.[2] Recall that R is the set of r registers that were written to in execution E_r. For each register in R, let a new process clone the process that last wrote to it all the way to covering the register poised to write the same value as present in the register in configuration C_r.

Let us now apply the covering argument utilizing the clones. Consider execution $E_r\alpha\gamma\beta$, where γ is a block write to R by the new clones. We know that process p returns 0 after $E_r\alpha$. During its solo execution α, process p has to write to a register outside of R. Otherwise, the configuration after $E_r\alpha\gamma$ is indistinguishable from C_r to process q as the values in all registers are the same, and q is still in the same state as in C_r. Hence, q will return 1 after $E_r\alpha\gamma\beta$ as it would after $E_r\beta$, contradicting the correctness of the consensus algorithm. Analogously, process q has to write outside of R during β. Let $\alpha = \alpha' w_p \alpha''$, where w_p is the first write of p outside the set of registers R, and let $\beta = \beta' w_q \beta''$, with w_q being the first write outside of R. Let ℓ be the length of $\gamma\beta' w_q$ and B_i be a prefix of $\gamma\beta' w_q$ of length i, for all possible $0 \leq i \leq \ell$.

Next, we use a valency argument to reach C_{r+1}. We show that either the configuration reached after $\mathsf{E}_r\alpha'\gamma\beta' w_q$, or one of the configurations reached after $\mathsf{E}_r\alpha' B_i w_p$ for some i, satisfies the properties necessary to be C_{r+1}. Clearly, we have used the right number of processes to reach any of these configurations and $r+1$ registers have been written to while doing so, including R and the register written by w_p or w_q. Thus, we only need to show that one of these configurations is *bivalent*.

Assume the contrary. The configuration for $i = 0$ must be 0-univalent, since p returns 0 only throughout α'', and we assumed that the configuration is not bivalent. Similarly, the configuration reached after $\mathsf{E}_r\alpha'\gamma\beta' w_q = E_r\alpha' B_\ell$ is 1-univalent. It is univalent by our assumption and 1-valent as q running solo returns 1 through β'' (α' does not involve a write outside of R and q cannot distinguish from $E_r\beta' w_q \beta''$). Because the configuration reached after $E_r\alpha' B_\ell$ is

[2] Alternatively one can say execution $E_r\alpha$ ends with p returning 0 and $E_r\beta$ ends with q returning 1.

1-univalent, any terminating solo execution of process p from that configuration must also return 1. In particular, every terminating solo execution that starts by p performing its next step w_p returns 1. So the configuration reached after $E_r\alpha' B_\ell w_p$ must be 1-univalent: solo executions of p return 1 (some solo execution terminates due to nondeterministic solo execution), and it is univalent by our assumption (it is the same as configuration for $i = \ell$). Therefore, the configuration reached after $\mathsf{E}_r\alpha' B_i w_p$ is 0-univalent for $i = 0$ and 1-univalent for $i = \ell$. Hence, we can find a switching point for some i and $i + 1$, where the configuration X reached by $\mathsf{E}_r\alpha' B_i w_p$ is 0-univalent, while the configuration Y reached by $\mathsf{E}_r\alpha' B_{i+1} w_p$ is 1-univalent. Let o be the extra operation in B_{i+1}.

Operation o is not by p and may not be a read or a write to the same register as w_p writes to since p would not distinguish between X and Y and would return the same output from both configurations through the same solo execution, contradicting the existence of the different univalencies. Otherwise, operations w_p and o commute. Let σ be a terminating solo execution from Y by the process that performed operation o, where it returns 1 due to the univalency of Y. Also consider this process performing its next operation o from X. Since w_p and o commute, and o is not a read, the process cannot distinguish between the resulting configuration and Y and returns 1 through σ as from Y. However, $o\sigma$ is a solo execution from X that returns 1, contradicting the 0-univalency of X. The contradiction proves the induction step, completing our induction.

Notice that for n processes, Lemma 1 directly implies the existence of an execution where $\Omega(\sqrt{n})$ registers are written to, proving the desired lower bound.

2.2 Linear Lower Bound

Consider systems with n anonymous processes and an arbitrary correct consensus algorithm satisfying the nondeterministic solo termination property. We will assume that no execution of the algorithm uses more than $n/20$ registers (otherwise, we are trivially done), and prove that such an algorithm has to use $\Omega(n)$ registers, which completes the proof. For notational convenience, let us define m to be $n/20$.

The argument in Lemma 1 relies on a new set of clones in each iteration to overwrite the changes to the contents of the registers made during the inductive step. This is the primary reason why we only get an $\Omega(\sqrt{n})$ lower bound. As the authors of [FHS98] also mention, to get a stronger lower bound we would instead have to reuse existing processes. In order to do so, these existing processes need to cover the registers in our inductive configurations (we must also ensure proper valency conditions on what they are about to write, but let us focus on the covering). Now, even if we reach such a configuration, during a solo execution interval of some process in the subsequent induction step, all the registers may get written to, and we would have to use all the covering existing processes to overwrite the changes. Therefore, in the next configuration, there is no way to guarantee that the existing processes would still cover various registers.

This is the primary reason why we have to replace solo executions in the proof with a different class of executions that we call *reserving*. Intuitively, reserving executions ensure that for the registers that are written to, some processes are reserved to cover them. This way, we can have reserved processes cover the registers in subsequent inductive configurations. Notice that the definition of valency used in the proof of Lemma 1 was based on solo executions. Thus, we also redefine valency based on reserving executions.

Reserving executions The following is a formal definition of a reserving execution interval.

Definition 1. *Let C be some configuration reachable by the algorithm, and let P be a set of at least $m+1$ processes. We call an execution interval γ that starts from configuration C* reserving *from C by P if:*

- *Every step in γ is by a process in P.*
- *At any time during the execution of γ: if we let R_w be the set of registers written to so far during γ, then, for each register in R_w, there is a* reserved *process $p \in P$ covering that register, one per register.*
- *If a process $p \in P$ returns during γ then it does so in the last step of γ.*

Notice that by definition any prefix of a reserving execution interval is also a reserving execution interval. Let $\mathsf{Res}(C, P)$ be the set of all reserving execution intervals from C by processes in P that end with a process $p \in P$ returning. We first show that given sufficiently many processes, such an execution interval exists. This is essential for defining the valency later. Recall that we assumed a strict upper bound of m on the number of registers that can ever be written.

Lemma 2. *For any reachable configuration C and a set of at least $m+1$ processes P, none of which have returned yet, we have that $\mathsf{Res}(C, P) \neq \varnothing$.*

Proof. For a given C and P, we will prove the lemma by constructing a particular reserving execution interval γ that ends when some process $p \in P$ returns. We start with an empty γ and continuously extend it. In the first stage, one by one, for each process $p \in P$:

- Due to the nondeterministic solo termination, there exists a solo execution of p where p returns.
  - If p ever writes to any register during this solo execution, extend γ by the prefix of the execution before this write, and move to the next process in P.
  - Otherwise, complete γ by extending it with the whole solo execution of p.

We have finitely many processes and the first stage described above consists of extending the execution interval at most $|P|$ times. Each time, because of the nondeterministic solo termination for some process $p \in P$, we extend γ by a prefix of a finite solo execution of p. Moreover, all operations are reads by processes in P, and therefore the prefix of γ constructed so far is reserving.

If some process returns in the first stage, the construction of γ is complete. Otherwise, since the first stage is finite, we move on to the second stage described below. In the configuration after the first stage each of the at least $m+1$ processes in P is covering a register (by their next write operation after the first stage). From that configuration, the execution interval γ is extended by repeatedly doing the following:

1. Let R be the set of covered registers by processes of P. Since $|R| \leq m < |P|$, we can find two processes $p, q \in P$ covering the same register in R.
2. Due to the nondeterministic solo termination, there exists a solo execution of p where p returns.
   - If p ever writes to a register outside of R during this solo execution, extend γ by the prefix of the execution before this write, and continue from the first step. Notice that at the beginning of the next iteration, process p still covers a register as required.
   - Otherwise, complete γ by extending it with the whole solo execution of p.

In the second stage, each iteration terminates, since for any process $p \in P$, we can extend by at most the terminating solo execution of p, which exists and is finite. After each iteration, if the construction is not complete, the size of R increases by one. But there are at most m registers in the system and $|R| \leq m$. Thus, after at most m finite extensions, we will complete the construction of γ when some process returns.

The execution is reserving because at all times, the registers that were written-to are in R. Moreover, for each register in R, there is always a process covering it starting from the time it was first covered by some process p in the second step of some iteration all the way until the end of γ.

The next lemma follows immediately from the definition of reserving executions.

Lemma 3. *Consider a reachable configuration C, a set of at least $m+1$ processes P' none of which have returned yet, and another configuration C' reached after some process $p \notin P'$ performs a write operation w_p in C. Moreover, assume that another process $q \neq p$ with $q \notin P'$ is covering the same register that w_p writes to. Then if $\gamma \in \mathsf{Res}(C', P')$, then $w_p\gamma$ is in $\mathsf{Res}(C, P)$ where $P = P' \cup \{p\} \cup \{q\}$.*

New definition of valency We say that a configuration C is 0-valent$_U$ with respect to the set of processes U, if there exists a subset of at least $m+1$ processes $P \subseteq U$ and a reserving execution in $\mathsf{Res}(C, P)$ that finishes when some process in P returns 0. We call C 0-valent$_U^{m+1}$ w.r.t. U, if there exists a subset of *exactly* $m+1$ processes $P \subseteq U$ ($|P| = m+1$), and a reserving execution interval in $\mathsf{Res}(C, P)$ returning 0. We define 1-valent$_U$ and 1-valent$_U^{m+1}$ analogously. If U contains at least $m+1$ processes that have not returned, Lemma 2 implies that every configuration is 0-valent$_U^{m+1}$ or 1-valent$_U^{m+1}$ (and thus 0-valent$_U$ or 1-valent$_U$).

As in our earlier definition in Section 2.1, but unlike the standard definition, a configuration that is 0-valent$_U^{m+1}$ can still also be 1-valent$_U^{m+1}$ in which case we call it bivalent$_U^{m+1}$. Basically, a configuration is bivalent$_U^{m+1}$ if it is both 0-valent$_U^{m+1}$ due to some $P \subseteq U$ and 1-valent$_U^{m+1}$ due to some $Q \subseteq U$. A configuration that is not bivalent$_U^{m+1}$ is called univalent$_U^{m+1}$. Finally, similar to our earlier convention, we define a configuration to be 0-univalent$_U^{m+1}$ if it is 0-valent$_U^{m+1}$ but not 1-valent$_U^{m+1}$. On the other hand, a configuration that is 1-valent$_U^{m+1}$ but not 0-valent$_U^{m+1}$ is called 1-univalent$_U^{m+1}$. Terms bivalent$_U$, univalent$_U$, 0-univalent$_U$ and 1-univalent$_U$ are defined analogously.

Next we prove a lemma that lets us find reserving executions consisting of disjoint processes.

Lemma 4. *Consider a configuration C which is bivalent$_U$ w.r.t. U. Assume that there are (possibly intersecting) sets of at least $m+1$ processes each $P \subseteq U$ and $Q \subseteq U$ such that $|U| \geq |P|+|Q|+m$, and some reserving execution in $\mathsf{Res}(C, P)$ ends when $p \in P$ returns 0, while some reserving execution in $\mathsf{Res}(C, Q)$ ends when $q \in Q$ returns 1. Then there are also disjoint sets of processes $P' \subseteq U$ and $Q' \subseteq U$ $(P' \cap Q' = \varnothing)$, such that an execution in $\mathsf{Res}(C, P')$ returns 0 and an execution in $\mathsf{Res}(C, Q')$ returns 1. Moreover, $m+1 \leq \min(|P'|, |Q'|) \leq \min(|P|, |Q|)$ and $\max(|P'|, |Q'|) \leq \max(|P|, |Q|)$.*

Proof. None of the processes in U may have already returned in configuration C, as that would contradict the existence of a reserving execution returning the other output. If P and Q do not intersect then we set $P' = P$ and $Q' = Q$. Otherwise, we can find a set $H \subseteq U - P - Q$ of $m+1$ processes. By Lemma 2, $\mathsf{Res}(C, H)$ is non-empty, and without loss of generality, some execution in $\mathsf{Res}(C, H)$ returns 0. Then, we set $P' = H$ and $Q' = Q$ (if all executions in $\mathsf{Res}(C, H)$ return 1, we would set $P' = P$ and $Q' = H$).

The process-clone pairs and the proof As mentioned earlier, it is obviously not sufficient to simply cover registers with existing processes without any knowledge of what they are about to write. In the proof of Lemma 1 we used new clones that covered registers to block-overwrite these registers back to the contents whose valency we knew. In order to do something similar with existing processes, we associate a dedicated clone to each process. The process and its clone remain in the same states and perform the same operations during the whole execution.

Usually, when we schedule a process to perform an operation, its clone performs the same operation immediately after the process. Thus the pair of the process and the clone remain in the same state. Under these circumstances, we can treat the pair of the process and its clone as a single process, because no process can distinguish the execution from when the clone would not take steps. However, sometimes we will *split* the pair by having only the process perform a write operation and let the clone cover the register. We will explicitly say when this is the case. After we split the pair of process and clone in such a way, we

will not schedule the process to take any more steps and thus the clone will remain poised to write to the covered register. After some delay, we will schedule the clone of the process to write, effectively resetting the register to the value it had when the process wrote. Moreover, because meanwhile the process did not take any steps, after the write the clone will again be in the same state as its associated process. Hence the pair of the process and clone will no longer be split, and will continue taking steps in sync like a single process.

This is different from the way clones were used in the proof of Lemma 1, because after the pair of the process and its clone is united, it can be split again. Therefore, the same clone can reset the contents of registers written by its associated process multiple times, instead of requiring a new clone every time.

We call a split pair of a process and a clone *fresh* as long as the register that the process wrote to, and its clone is covering, has not been overwritten. After the register is overwritten, we call the split pair *stale*.

In addition, we also use cloning in a way similar to the proof of Lemma 1, except that we do this at most constantly many times, as opposed to r times, to reach the next configuration C_{r+1}. Moreover, each time when we do this, we create duplicates of both the process and its corresponding clone. This new process-clone pair is in the same state as the original pair, and from there on behaves like a single new process similar to all other pairs. We will always consider valency with respect to sets of processes whose pairs are not split. Therefore, the definition of valency does not need to change when the clones keep taking steps immediately after their processes.

Sometimes, when considering process-clone pairs, none of which are split, we may refer to them as processes, i.e. we may talk about a process taking steps or returning a value. As mentioned earlier, it is assumed that as long as the pair is not split, the clone always follows and takes the same steps right after the process. Hence, in this context, a process taking a step means a pair taking a step.

Now we are ready to prove the main result.

Theorem 1. *In the system of anonymous processes, consider any correct consensus algorithm satisfying nondeterministic solo termination, with the property that every execution uses at most m registers. For each r with $0 \leq r \leq m$, there exists a set U containing $5m+6+2r$ process-clone pairs such that a configuration C_r is reachable through an execution E_r by processes and clones in U with the following properties:*

1. *There exists a set R of r registers, that can be partitioned in two disjoint subsets $R = R_s \cup R_c$, where:*
   - *R_s consists of all registers in the system that each have one fresh split pair on them, last written by some process whose clone has not yet performed the write and is covering the register.*
   - *$R_c = R - R_s$. Each register in R_c is covered by an unique pair of both a process and its clone.*

Thus, each fresh pair is split on a different register in R_s, and an additional $|R_c|$ pairs are covering the registers in R_c. Let V be the set of these $|R_s| + |R_c| = r$ pairs.

2. *There are at most r stale split pairs in the system, that are all split on pairwise different registers from R. Let L be the set of these at most r stale split pairs.*
3. *There exist disjoint sets of process-clone pairs that are not split $P, Q \subseteq U - V - L$ with $|P| + |Q| \leq 2m + 4$, such that an execution in $\mathsf{Res}(C_r, P)$ returns 0 and an execution in $\mathsf{Res}(C_r, Q)$ returns 1.*[3]

Proof. The proof is by induction on r, with the base case $r = 0$. Out of the $5m + 6$ processes-clone pairs, half of them start with an input 0 and half start with an input 1. We let C_0 be the initial state, P be a set of some $m + 1$ pairs with input 0, and Q be a set of some $m + 1$ pairs with input 1. The first two properties are trivially satisfied; also $P \cap Q = \varnothing$ and $|P| + |Q| = 2m + 2$. By Lemma 2 and correctness of consensus, there is a reserving execution in $\mathsf{Res}(C_0, P)$ that decides 0, and a reserving execution in $\mathsf{Res}(C_0, Q)$ that decides 1 (C_0 is bivalent$_U$). Observe that the pairs are not split and for the purposes of valency we can just consider the steps of processes.

Now, let us assume induction hypothesis for some r, i.e. the existence of E_r and C_r with the required three properties, and prove the step for $r + 1$ by extending E_r to E_{r+1}, resulting in the configuration C_{r+1}. Let U, P, Q, V, L and $R = R_s \cup R_c$ all be defined as in the theorem statement for r. Our goal is to construct sets U', P', Q', V', L' and $R' = R'_s \cup R'_c$ for $r + 1$. In $U' - U$ we have two more process-clone pairs available that have not taken steps and can be used to clone an existing process-clone pair. Let T denote $U - V - L - P - Q$. Since $|V| = r$, $L \leq r$ and $|P| + |Q| \leq 2m + 4$, we have $|T| \geq 3m + 2$.

For all but $|R_s| + |L|$ split pairs both processes and clones are in the same states, about to perform the same operations. By definition, each stale pair in L is split on a different register from R. In the following argument, we extend the execution from E_r to E_{r+1} by steps of processes and clones not in L. This can introduce new stale split pairs and the resulting configuration C_{r+1} may not immediately satisfy the second property. We will then show how to modify the extension and unite some stale split pairs, such that the resulting configuration satisfies all properties, including the second property with the new L'.

Let $\alpha \in \mathsf{Res}(C_r, P)$ be the reserving execution interval that returns 0, and let $\beta \in \mathsf{Res}(C_r, Q)$ be the reserving execution interval that returns 1. Notice that each time a process in P or Q takes a step in α or β, its clone performs an identical step immediately after. The execution $E_r\alpha$ ends with a process-clone pair $p \in P$ returning 0 and the execution $E_r\beta$ ends with a process-clone pair $q \in Q$ returning 1.

Each register in R_c was covered by some pair of both a process and its clone in V. Let γ_c be a block write to all registers in R_c by only the processes but not the

[3] The pairs of processes in P and Q are not split, because all split pairs belong to $V \cup L$ (fresh to V and stale to L). Also, the third condition implies that the configuration C_r is bivalent$_{U-V-L}$.

clones of these respective covering pairs: i.e. after each write we get a new fresh split pair. Consider a configuration D reached from C_r by executing this block write, i.e. a configuration reached after $E_r\gamma_c$. Assume that D is 1-valent$_T^{m+1}$, without loss of generality, because it has a valency. For any execution interval e, let us denote by $W(e)$ the set of registers written to during e. Hence, $R_s \cap W(e)$ is the set of registers in R_s that are written-to during e. Each register in R_s is covered by a clone of a split pair whose process has already performed the write and is stopped. Define $\gamma_s(e)$ as a block write to all registers in $R_s \cap W(e)$ by these trailing clones of the split pairs in V: i.e. after each write another clone catches up with its process and a previously split pair is united. Basically, if we run an execution interval e from C_r that changes contents of some registers in R_s, we can then clean these changes up by executing $\gamma_s(e)$, which leads to all registers in R_s having the same contents as in C_r.

Using a crude covering argument we can first show that

Lemma 5. *The execution interval α must contain a write operation outside R.*

The proof of this lemma is provided later.

Based on this we can write $\alpha = \alpha' w_p \alpha''$, where w_p is the write operation to a register $\mathsf{reg} \notin R$, performed by some process-clone pair $p \in P$.

Looking ahead, when we reach C_{r+1}, the new set of registers R' will be $R \cup \{\mathsf{reg}\}$. Next, we prove the following lemma using an FLP-like case analysis:

Lemma 6. *We can extend execution E_r (i.e. from C_r) with an execution interval e and reach a configuration satisfying the first and the third inductive requirements to be C_{r+1} with a properly defined U', P', Q', V' and $R' = R'_s \cup R'_c$, and with all process-clone pairs that are not split being in sync. But the second property is not immediately satisfied. All stale split pairs from L remain stale and split, but some pairs that were fresh and split on registers in $R_s \cap W(e)$ may have become stale in C_{r+1} (because neither the process nor the clone in the split pair has taken steps while the register was overwritten in e). However, these are the only possible new stale split pairs in C_{r+1}, and they do not belong to the new sets $V' \cup P' \cup Q'$.*

The proof of this lemma can be found in the full version.[4]

In order to finish the proof of the theorem, we need to show how to construct L'. According to the above Lemma 6 we can extend the execution to reach the next configuration C_{r+1} satisfying first and third but not the second property about the stale split pairs L'. In C_r we had at most r stale pairs in the system, each split on a different register, and L was the set of these pairs. But on the way to reaching C_{r+1}, we may have introduced new stale pairs in the system. According to Lemma 6 these must be the pairs that were fresh and split on registers in $R_s \cap W(e)$ in C_r, and whose associated register in R_s has been overwritten during e, making them stale in C_{r+1}.

The set of all stale pairs in C_{r+1} may not satisfy the requirements imposed for L', since there could already have been a stale pair split on a register in

[4] Available at http://arxiv.org/abs/1506.06817.

$R_s \cap W(e)$ in L (in C_r). Then two stale pairs would be split on this register in C_{r+1}, violating the second property. However, for each such register in $R_s \cap W(e)$, we know a stale pair $\rho \in L$ was split on it in C_r, and that this register was written-to during extension $W(e)$. We now modify the extension e; we add a single write by the clone of the stale split pair ρ immediately before a write operation to the same register that was already in e. This way, no pair other than the clone of ρ observes a difference between the two executions, and we will use the configuration reached by the modified execution as C_{r+1}. Because of this indistinguishability, the new C_{r+1} still satisfies other required properties. Moreover, the pair ρ is not split anymore; it is united since the clone has caught up with its process.

We can do the above modification to the execution for each register in $R_s \cap W(e)$ that previously ended up with two stale split processes in C_{r+1}. Let the modified execution extension be e'. In e', some stale split pairs from L are united, indistinguishably to all other processes and clones, leading to a configuration C_{r+1}, that still satisfies the first and third properties, and has at most one stale pair split on any register. We take L' to be the set of stale split pairs. By construction, all stale pairs are split on registers in R' and no two on the same register, so we do have $|L'| \leq r + 1$ as desired. Hence, we have reached configuration C_{r+1} satisfying all properties and completing the proof.

Corollary 1. *In a system of n anonymous processes, any consensus algorithm satisfying non-deterministic solo termination must use $\Omega(n)$ registers.*

Proof. Theorem 1 directly implies the $\Omega(n)$ lower bound on the number of registers used in some execution. If n is the number of anonymous processes and no execution uses more than $m = n/20$ registers, by Theorem 1 we can reach C_m for large enough n, and we have enough processes $n \geq 10m + 12 + 4m$. In C_m there are m registers in R, each of which has either already been written-to (R_s) or are covered by unique processes (R_c). We could perform a block write to R_c by covering processes from V in C_m, after which in the resulting execution $m = n/20 = \Omega(n)$ different registers would have been written to.

We now provide the delayed proof of Lemma 5.

Proof. Assume the contrary. We know that the execution $E_r\alpha$ decides 0. No process or clone that takes a step in γ_c or $\gamma_s(\alpha)$ appears in α (they belong to V, disjoint from P and Q), and by definition, no process or clone from T takes a step in α, γ_c or $\gamma_s(\alpha)$. Thus, to all processes (and clones) in T, the configurations after $E_r\alpha\gamma_s(\alpha)\gamma_c$ and after $E_r\gamma_c$, which is configuration D, are indistinguishable. This is because no process (or clone) in T has taken steps, the registers in R contain the same values, and other registers were not touched during α, $\gamma_s(\alpha)$ or γ_c. Configuration D is 1-valent$_T^{m+1}$, so some extension from $E_r\alpha\gamma_s(\alpha)\gamma_c$ by an execution interval from $\mathsf{Res}(D, T)$ decides 1. This contradicts the correctness of the algorithm.

3 Extensions

Adaptive Lower Bound: Let us sketch a proof for an adaptive linear lower bound on the space complexity of consensus for non-anonymous processes but under extra restrictions on register size and solo termination. In this setting, processes are no longer anonymous, but we assume they come from a very large namespace. Each of these huge number of processes executes its own code, however, we get to choose which subset of processes participates in the execution. We show that there is a linear space lower bound that depends on the number of participating processes, that is, for large enough namespace, we can find an execution of n processes (out of all processes) where $\Omega(n)$ registers get written.

The restrictions are that the registers have a bounded size and that the consensus algorithm satisfies bounded nondeterministic solo termination property, meaning that there always is a terminating solo execution of a process consisting of less than certain number of steps. If we had bounded nondeterministic solo termination, the lower bound execution for anonymous processes constructed in Theorem 1 would always contain less than B steps, where B is a finite bound that only depends on n and the solo termination bound. As registers have a bounded size, for both input values, a process can exhibit only finitely many different behaviors during its first B steps, because in each step it can either read or write a fixed number of different values. For a sufficiently large namespace (depending on B, n and register size), by pigeon-hole principle, we can find n processes such that half of them start with input 1, half start with 0 and all processes with the same input behave as anonymous for the first B steps of an execution. Hence, we can use Theorem 1 and get an execution where $n/20$ registers are written to, as described at the end of Section 2.2.

Future Work: We believe that is should be possible to derive the above adaptive lower bound without the bounded solo termination assumption, and to get good estimate on the required size of the namespace. However, the major open problem is still to resolve the general, non-anonymous and non-adaptive case, i.e. to get tight bounds on the space required to solve consensus with exactly n asymmetric processes.

4 Acknowledgments

Support is gratefully acknowledged from the National Science Foundation under grants CCF-1217921, CCF-1301926, and IIS-1447786, the Department of Energy under grant ER26116/DE-SC0008923, and the Oracle and Intel corporations.

The author would like to thank Nir Shavit, Michael Coulombe and Dan Alistarh for helpful conversations and feedback, and the anonymous reviewers for their excellent comments.

References

AE14. Hagit Attiya and Faith Ellen. Impossibility results for distributed computing. *Synthesis Lectures on Distributed Computing Theory*, 5(1):1–162, 2014.

AH90. James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, September 1990.

BO83. Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, pages 27–30, New York, NY, USA, 1983. ACM.

FHS98. Faith Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *Journal of the ACM (JACM)*, 45(5):843–862, 1998.

FLP85. Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

GHHW13. George Giakkoupis, Maryam Helmi, Lisa Higham, and Philipp Woelfel. An $\mathcal{O}(\sqrt{n})$ space bound for obstruction-free leader election. In *Distributed Computing*, pages 46–60. Springer, 2013.

GHHW14. George Giakkoupis, Maryam Helmi, Lisa Higham, and Philipp Woelfel. Test-and-set in optimal space. 2014. Accepted to STOC 2015.

GR05. Rachid Guerraoui and Eric Ruppert. What can be implemented anonymously? In *Distributed Computing*, pages 244–259. Springer, 2005.

GW12. George Giakkoupis and Philipp Woelfel. On the time and space complexity of randomized test-and-set. In *Proceedings of the 2012 ACM symposium on Principles of Distributed Computing*, pages 19–28. ACM, 2012.

SP89. Eugene Styer and Gary L Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 177–191. ACM, 1989.