

Tight Bounds for Adopt-Commit Objects

James Aspnes* Faith Ellen†

November 21, 2011

Abstract

We give matching upper and lower bounds of $\Theta\left(\min\left(\frac{\log m}{\log \log m}, n\right)\right)$ for the space and individual step complexity of a wait-free m -valued adopt-commit object implemented using multi-writer registers for n anonymous processes. While the upper bound is deterministic, the lower bound holds for randomized adopt-commit objects as well. Our results are based on showing that adopt-commit objects are equivalent, up to small additive constants, to a simpler class of objects that we call conflict detectors.

Our anonymous lower bound also applies to the individual step complexity of m -valued wait-free anonymous consensus, even for randomized algorithms with global coins against an oblivious adversary. The upper bound can be used to slightly improve the cost of randomized consensus in an oblivious-adversary model.

For *non-anonymous* deterministic implementations of adopt-commit objects, we show a lower bound of $\Omega\left(\min\left(\frac{\log m}{\log \log m}, \frac{\sqrt{\log n}}{\log \log n}\right)\right)$ and an upper bound of $O\left(\min\left(\frac{\log m}{\log \log m}, \log n\right)\right)$ on the worst-case individual step complexity.

1 Introduction

An **adopt-commit object** [2, 20] or **ratifier** [3] is a one-shot shared-memory object that represents the **adopt-commit protocols** of [15] and can be used to implement round-based protocols for set-agreement and consensus. An m -valued adopt-commit object supports a single operation, **adoptCommit**(u), where u is an input from a set of m **values**. The result of this operation is an output of the form (**commit**, v) or (**adopt**, v), where the second component is a value from this set and the first component is a **decision bit** that indicates whether the process should decide value v immediately or adopt it as its preferred value in later rounds of the protocol. Improving the performance of adopt-commit objects can improve the performance of consensus protocols that use them.

*Yale University, Department of Computer Science. Supported in part by NSF grant CCF-0916389.

†University of Toronto, Department of Computer Science. Supported in part by the Natural Science and Engineering Research Council of Canada.

Lower bounds on adopt-commit objects also yield immediate lower bounds on consensus.

The requirements for an adopt-commit object are:

1. **Validity.** The output value of an operation is the input of some (possibly different) operation.
2. **Termination.** With probability 1, each nonfaulty process produces an output in a finite number of steps, where the probability is taken over the coin tosses performed by the algorithm.
3. **Coherence.**¹ If the output of some operation is (commit, v) , then every output is either (adopt, v) or (commit, v) .
4. **Convergence.** If all inputs are v , all outputs are (commit, v) .

These requirements are closely related to the validity, termination, and agreement requirements for consensus. The difference is that agreement (which requires that all outputs are the same) is replaced by the weaker requirements of coherence and convergence. As observed in [3], this means that consensus objects satisfy the requirements of adopt-commit objects, if each process returns the decision bit **commit** together with its output value. It follows that lower bounds on adopt-commit objects immediately give lower bounds on consensus objects.

Until now, the best implementations of m -valued adopt-commit objects had $\Theta(n)$ individual step complexity, for n processes [15], or $\Theta(\log m)$ individual step complexity, for any number of processes [3]. Both these implementations are deterministic, but the latter is also **anonymous**. This means that all processes run the same code. Differences between the behaviour of two different processes can arise only as a result of different input values, (different supplies of random bits, in the case of a randomized protocol), and when they are scheduled. A number of advantages of anonymity are discussed in [5].

Here, we consider how much further we can improve the complexity of an implementation of an adopt-commit object without losing anonymity. We give two simple, deterministic, anonymous protocols for detecting multiple input values, from which we obtain implementations of m -valued adopt-commit objects. One of these has $O(n)$ individual step complexity, given an upper bound, n , on the number of processes. The other has $O\left(\frac{\log m}{\log \log m}\right)$ individual step complexity, for any number of processes. While this is only a small improvement in complexity, we show a matching lower bound on the individual step complexity of any anonymous implementation (including randomized implementations against an oblivious adversary) of an m -valued adopt-commit object that supports at least $\Omega\left(\frac{\log m}{\log \log m}\right)$ processes. Our lower bound also implies a lower

¹The definition of adopt-commit objects in [2] uses the term *agreement* for this property. We use *agreement* instead for the stronger unconditional agreement property of consensus objects. The term *coherence* is from [3].

bound of $\Omega\left(\frac{\log m}{\log \log m}\right)$ on the individual step complexity for anonymous randomized consensus with sufficiently many processes, even against an oblivious adversary.

We can extend our anonymous bounds to give a partial characterization of the worst-case individual step complexity for non-anonymous adopt-commit objects. We show that *deterministic* non-anonymous implementations have a worst-case individual step complexity between $\Omega\left(\min\left(\frac{\log m}{\log \log m}, \frac{\sqrt{\log n}}{\log \log n}\right)\right)$ and $O\left(\min\left(\frac{\log m}{\log \log m}, \log n\right)\right)$.

2 Conflict detectors

We introduce a simpler object, a conflict detector, and show that it can be implemented from an adopt-commit object. We also show that a conflict detector and registers can be used to implement an adopt-commit object.

An m -valued **conflict detector** supports a single operation, **check**(v), with input v from a set of m values. It returns **true** (to indicate a conflict) or **false** (to indicate no conflicts), and has the following two properties:

- In any execution that contains a **check**(v) operation and a **check**(v') operation with $v \neq v'$, at least one of these two operations returns **true**.
- In any execution in which all **check** operations have the same input value, they all return **false**.

2.1 Non-linearizability

Neither adopt-commit objects nor conflict detectors are required to be linearizable. An implementation of an object is **linearizable** [17] if, for any execution in which processes perform operations on the implemented object, there is a sequential execution with the same operations such that each operation returns the same response in both executions and non-concurrent operations in the original execution occur in the same order as in the sequential execution.

It is easy to show that neither adopt-commit objects nor conflict detectors have deterministic linearizable implementations from atomic registers, and our implementations are not linearizable. A solo execution of **adoptCommit**(v) must return (**commit**, v), since it is possible that no process has a conflicting input. Hence, if the first operation linearized in an execution is **adoptCommit**(v), then it must return (**commit**, v), and the output value of every subsequent **adoptCommit** operation in the execution must be v . Hence a deterministic linearizable implementation of an adopt-commit object gives rise to a deterministic implementation of consensus, which does not exist [14, 18]. Similarly, the first **check** operation linearized in any execution of a conflict detector must return **false** and subsequent **check** operations with different inputs must return **true**, which can be used to implement test-and-set, for which no deterministic implementation from registers exists [16].

2.2 Equivalence of adopt-commit objects and conflict detectors

In this section, we show that the individual step complexities of adopt-commit objects and conflict detectors differ by small additive constants. Because our reductions are anonymous, this also holds for anonymous implementations. We use T_{ac} and T_{ch} to denote the worst case step complexities of the `adoptCommit` and `check` operations, respectively.

We begin by giving a straightforward implementation of a conflict detector from an adopt-commit object. The code is presented in Algorithm 1.

```

shared data:
  adopt-commit object  $r$ .
1 procedure check( $v$ )
2 begin
3    $(d, v') \leftarrow r.\text{adoptCommit}(v)$ 
4   if  $(d, v') = (\text{commit}, v)$  then
5     return false
6   else
7     return true
8   end
9 end

```

Algorithm 1: A conflict detector using an adopt-commit object.

Lemma 1 *Algorithm 1 implements a conflict detector with $T_{ch} = T_{ac}$.*

Proof: If all `check` operations have the same input v , then, they all call $r.\text{adoptCommit}(v)$, which, by the convergence property, all return (commit, v) . Consequently, all the `check`(v) operations return **false**.

If there is an execution in which two operations, `check`(v) and `check`(v') are performed, where $v \neq v'$, then they call $r.\text{adoptCommit}(v)$ and $r.\text{adoptCommit}(v')$, respectively. By coherence, either the result of $r.\text{adoptCommit}(v)$ is not (commit, v) or the result of $r.\text{adoptCommit}(v')$ is not (commit, v') . It follows that **true** is returned by at least one of these two `check` operations. Thus, Algorithm 1 implements a conflict detector.

The step complexity of `check` is the same as the step complexity of `adoptCommit`, since only line 3 contains a nonlocal operation. ■

Conversely, an adopt-commit object can be implemented from a conflict detector and two registers. One of the registers, `conflict`, contains a single bit. It is initially **false** and is set to **true** when a process detects a conflict. It is never reset to **false**. Each process reads `conflict` immediately before returning and uses its value to determine its decision bit. The other register, `proposal`, initially contains the special value \perp , which is different from all input values. A process writes its input value to `proposal` after reading it and seeing that it contains \perp . The code is presented in Algorithm 2.

shared data:

register **conflict**, initially **false**;

register **proposal**, initially \perp ;

conflict detector c .

```
1 procedure adoptCommit( $v$ ) begin
2   if  $c.check(v) = \mathbf{true}$  then
3      $\mathbf{conflict} \leftarrow \mathbf{true}$ 
4   end
5    $u \leftarrow \mathbf{proposal}$ 
6   if  $u = \perp$  then
7      $\mathbf{proposal} \leftarrow v$ 
8   else
9      $v \leftarrow u$ 
10  end
11  if  $\mathbf{conflict} = \mathbf{true}$  then
12    return ( $\mathbf{adopt}, v$ )
13  else
14    return ( $\mathbf{commit}, v$ )
15  end
16 end
```

Algorithm 2: An adopt-commit object using a conflict detector and registers.

Lemma 2 *Algorithm 2 implements an adopt-commit object with $T_{ac} = T_{ch} + 4$.*

Proof: The output value of an `adoptCommit` operation is its input value, if the process performing the operation read that `proposal` has value \perp on line 5. Otherwise, the value it read from `proposal` is output. Since the only values written to `proposal` are input values, validity is satisfied.

If all inputs have the same value, then `check` always returns **false**, so `conflict` remains **false**. Thus, the decision bit of every output is **commit** and convergence is satisfied.

If some `adoptCommit` operation returns (commit, v) , then the process, p , performing this operation either read v from `proposal` on line 5 or it wrote v to `proposal` on line 7. It also read that `conflict` = **false** on line 11. In particular, this means that p did not perform line 3 and, hence, received **false** from `check`. Thus, all processes that perform `check` with input values different from p 's receive **true** and will set `conflict` to **true** on line 3 before reading `proposal` on line 5. Since none of these processes read \perp from `proposal`, none of them change the value of `proposal`, so they can only read v from `proposal` and return (adopt, v) . Since v has been written to `proposal`, it follows that p 's input value is v . All processes with input value v , including p , can only read \perp or v from `proposal` and, hence, they can only output (commit, v) or (adopt, v) . It follows that coherence is satisfied.

Reads and writes to shared registers occur only on lines 3, 5, 7, and 11. Thus, the step complexity of `adoptCommit` is 4 greater than the step complexity of `check`. ■

The simplicity of a conflict detector makes it easier to obtain bounds on its step complexity. In the next two sections, we obtain asymptotically matching upper and lower bounds for anonymous implementations of conflict detectors. These results imply the same bounds for anonymous adopt-commit objects.

3 Upper bounds on anonymous conflict detectors

In this section, we give two complementary implementations of anonymous m -valued conflict detectors. The first uses $O\left(\frac{\log m}{\log \log m}\right)$ steps for any number of processes, while the second uses $O(n)$ steps, for any value of m , where n is an upper bound on the number of processes. By choosing the first implementation when m is small and the second when m is large, we obtain a conflict detector that runs in $O\left(\min\left(\frac{\log m}{\log \log m}, n\right)\right)$ steps. We show this bound is optimal in Section 4.

3.1 A permutation-based conflict detector

In the natural algorithm for two values, a process performing `check(b)`, for $b \in \{0, 1\}$, writes to $R[b]$ and then checks $R[1 - b]$. Then, whichever of $R[0]$ or

$R[1]$ is written first will later be seen to have a non- \perp value by any process that writes to the other register, detecting the conflict.

Algorithm 3 is a generalization of this algorithm from $m = 2$ values to $m = k!$ values. Each of the $k!$ possible input values v is mapped to a distinct permutation $\pi_v : \{1, \dots, k\} \rightarrow \{1, \dots, k\}$. Then, for any two different input values, there exist two registers which function as in the natural two-valued algorithm. Algorithm 3 thus implements an anonymous, deterministic conflict detector for $m \leq k!$ values using at most $2k$ operations for $\mathbf{check}(v)$. As a function of m , this gives a worst-case individual step complexity of $2 \text{fact}^{-1}(m) = O\left(\frac{\log m}{\log \log m}\right)$, where $\text{fact}(k) = k!$ is the factorial function.

```

shared data:
  registers  $R[1..k]$ , initially  $\perp$ .
1 procedure  $\mathbf{check}(v)$ 
2 begin
3   for  $i \leftarrow 1..k$  do
4      $r \leftarrow R[\pi_v(i)]$ 
5     if  $r = \perp$  then
6        $R[\pi_v(i)] \leftarrow v$ 
7     else if  $r \neq v$  then
8       return true
9     end
10  end
11  return false
12 end

```

Algorithm 3: Permutation-based conflict detector for m values.

Lemma 3 *Algorithm 3 implements a conflict detector.*

Proof: If all calls to \mathbf{check} have the same input value v , then only v will be written to each register $R[i]$ and no process ever observes any value other than v or \perp . In this case, all operations correctly return **false**.

Now suppose there is an execution E in which two processes, p_u and $p_{u'}$, with different input values, u and u' , both return **false**. Then both processes read from all of the registers $R[1], \dots, R[k]$ and the values u and u' will both be written to all of the registers. Let $j, j' \in \{1, \dots, k\}$ be two indexes such that j occurs before j' in π_u , but j' occurs before j in $\pi_{u'}$. Without loss of generality, suppose that u is written to $R[j]$ before u' is written to $R[j']$ in E . Then, when $p_{u'}$ or any other process with value u' reads $R[j]$, it will not see \perp . This is because, before it reads $R[j]$, it either writes u' to $R[j']$ or reads u' from $R[j']$. This implies that no process writes u' to $R[j]$, which is a contradiction. ■

3.2 A collect-based conflict detector

Algorithm 4 is another implementation of a conflict detector. It places no limit on the number of distinct values m , but it works only when an upper bound, n , on the number of processes is known. The worst-case individual step complexity of a $\text{check}(v)$ operation in Algorithm 4 is $3n + 1$.

```
shared data:
  registers  $R[1..n]$ , initially  $\perp$ ;
  1-bit atomic register done, initially false.
1 procedure  $\text{check}(v)$ 
2 begin
3   for  $i \leftarrow 1..n$  do
4     if done then
5       break
6     else
7        $R[i] \leftarrow v$ 
8     end
9   end
10  done  $\leftarrow$  true
11  for  $i \leftarrow 1..n$  do
12    if  $R[i] \neq v$  then
13      return true
14    end
15  end
16  return false
17 end
```

Algorithm 4: A collect-based conflict detector for n processes.

The essential idea is that once some process finishes the first loop in $\text{check}(v)$ and sets **done** to **true**, each of the at most $n - 1$ other processes can write to at most one location in R before seeing **done** = **true** and leaving the loop. Because no process executes the collect in the second loop until **done** = **true**, any views obtained by two different processes in this loop can differ in at most $n - 1$ places. It follows that no two processes with different inputs can both see their own input in all n positions during the collect. Therefore, at least one of them will return **true**. If all calls to check have the same input, then only this input will appear in R , so all the calls will return **false**.

More formally, we have shown:

Lemma 4 *Algorithm 4 implements a conflict detector.*

4 A lower bound on anonymous conflict detectors

In this section, we show that any m -valued conflict detector for n anonymous processes has $\Omega\left(\min\left(\frac{\log m}{\log \log m}, n\right)\right)$ worst-case solo step complexity. Fix some anonymous, deterministic implementation of an m -valued conflict detector. For each input value v , we consider the solo execution E_v in which a process executes $\text{check}(v)$ starting from the initial configuration. Note that, because processes are deterministic and anonymous, the sequence of operations in E_v is fully determined by v .

Let k_v be the step complexity of E_v . Let W_v be the set of registers that a process writes to in E_v and let X_v be the set of registers that it reads from but does not write to. Let A_v be the permutation of $W_v \cup X_v$ arranged in the order in which the registers in W_v are first written and the registers in X_v are last read in E_v .

Lemma 5 *For all distinct input values u and v , if $k_u + k_v \leq n$, then there exist two registers $R_i, R_j \in (W_u \cup X_u) \cap (W_v \cup X_v)$ that occur in different orders in A_u and A_v .*

Proof: Suppose there are two input values $u \neq v$ such that $k_u + k_v \leq n$ and all registers $R_i, R_j \in (W_u \cup X_u) \cap (W_v \cup X_v)$ occur in the same order in A_u and A_v . We show that an adversary can construct an execution E involving $k_u + k_v \leq n$ processes that is indistinguishable from E_u to some process p_u performing $\text{check}(u)$ and indistinguishable from E_v to some other process p_v performing $\text{check}(v)$. In this execution, both p_u and p_v return **false**, violating the specification of a conflict detector.

For each $R_i \in W_u \cap (W_v \cup X_v)$, let $\sigma_{i,u}$ be the first write to R_i in E_u and, for each $R_i \in X_u \cap (W_v \cup X_v)$, let $\sigma_{i,u}$ be the last read from R_i in E_u . Let $S_u = \{\sigma_{i,u} \mid R_i \in (W_u \cup X_u) \cap (W_v \cup X_v)\}$. Define $\sigma_{i,v}$ and S_v analogously.

The adversary starts by constructing an interleaving E' of the operations in E_u and E_v . The operations in E_u appear in the same order in E' . The adversary schedules each read operation $\sigma_{i,v} \in S_v$ immediately before $\sigma_{i,u}$ and schedules each write operation $\sigma_{i,v} \in S_v$ immediately after $\sigma_{i,u}$. Note that, by assumption, the operations in S_v appear in the same order in E' as they do in E_v , namely, in the order the registers $R_i \in (W_u \cup X_u) \cap (W_v \cup X_v)$ they access occur in A_u and A_v .

If no operations in S_v occur between $\sigma_{i,v}$ and $\sigma_{j,v}$, then, in E' , the adversary arbitrarily interleaves the operations in E_v that occur strictly between $\sigma_{i,v}$ and $\sigma_{j,v}$ with the operations in E_u that occur strictly between $\sigma_{i,u}$ and $\sigma_{j,u}$. Likewise, the adversary arbitrarily interleaves the operations in E_v that occur before the first operation in S_v with the operations in E_u that occur before the first operation in S_u and the operations in E_v that occur after the last operation in S_v with the operations in E_u that occur after the last operation in S_u . Hence, the operations in E_v appear in the same order in E' .

The sequence of operations in E' is not necessarily a valid execution, because p_u may read a value written by p_v or p_v may read a value written by p_u . To prevent this, we add **clones**, as used in [13]. A **clone** of a process p is a process with the same input and code as p , which proceeds in lockstep with p , reading and writing the same values as p , until immediately before some write to a register. The adversary has the clone perform that write at some later point in the execution to ensure that the value p reads from that register is the same as the value p last wrote there. After performing its delayed write, a clone performs no further steps.

For each register $R_i \in W_u \cap W_v$, the adversary adds one clone of p_u to E' for each read of R_i by p_u after $\sigma_{i,v}$ and one clone of p_v to E' for each read of R_i by p_v after $\sigma_{i,v}$. Let E be the resulting execution.

If $R_i \in W_u \cap W_v$, then, by construction, any read of R_i by p_u in E after $\sigma_{i,v}$ sees the same value it saw in E_u , namely, the value it last wrote to R_i . Any read of R_i prior to $\sigma_{i,u}$ sees the initial value of R_i , since $\sigma_{i,u}$ and $\sigma_{i,v}$ are, by definition, the first writes to R_i by p_u and p_v in E' and, hence, E .

If $R_i \in X_u \cap W_v$, then all reads of R_i by p_u in E occur at or before $\sigma_{i,u}$ and, hence, see the initial value of R_i , as they do in E_u . This is because, in E , all writes to R_i by p_v occur at or after $\sigma_{i,v}$, which is after $\sigma_{i,u}$.

If $R_i \in (W_u \cup X_u) - W_v$, then p_v does not write to R_i in E , so all reads of R_i by p_u are the same as in E_u . Finally, if $R_i \notin W_u \cup X_u$, then p_u does not read R_i in E . Thus E_u and E are indistinguishable to p_u .

Similarly, E_v and E are indistinguishable to p_v . ■

The following combinatorial lemma allows us to bound m as a function of the step complexities, k_v , of the solo executions E_v . The proof is similar to Lubell's proof [19] of Sperner's Lemma.

Lemma 6 *Let $\{A_1, \dots, A_m\}$ be a set of finite sequences without repetition such that, for any two sequences A_i and A_j , there exist elements $x_{i,j}$ and $y_{i,j}$ that appear in different orders in A_i and A_j . Then $\sum_{i=1}^m \frac{1}{|A_i|} \leq 1$.*

Proof: Let $A = \bigcup_{i=1}^m A_i$ be the set of all elements appearing in any of the sequences A_1, \dots, A_m . Choose an ordering of A uniformly at random. Let X_i be the indicator variable that has value 1, if the ordering of the elements in A_i is consistent with this ordering, and has value 0, otherwise. Let $X = \sum_{i=1}^m X_i$.

Note that $X_i = 1$ implies that $X_j = 0$ for all $j \neq i$. This is because $x_{i,j}$ and $y_{i,j}$ appear in different orders in A_i and A_j . It follows that $X \leq 1$.

For each sequence A_i , the probability that it is consistent with the chosen ordering is exactly $\frac{1}{|A_i|!}$, so $E[X_i] = \frac{1}{|A_i|!}$. Hence $\sum_{i=1}^m \frac{1}{|A_i|!} = \sum_{i=1}^m E[X_i] = E[X] \leq 1$. ■

Theorem 7 *The worst-case solo step complexity of any anonymous deterministic implementation of an m -valued conflict detector for n processes is at least $\min(\text{fact}^{-1}(m), n/2)$, where $\text{fact}(\ell) = \ell!$ is the factorial function.*

Proof: Fix any anonymous deterministic implementation of an m -valued conflict detector for n processes and let k be its worst-case solo step complexity. Then, for every input value v , $|A_v| \leq k_v \leq k$.

If $k > n/2$, then the claim is true, so suppose that $k \leq n/2$. Then, for all distinct inputs u and v , $k_u + k_v \leq n$ and, hence, by Lemma 5, there are two registers that occur in different orders in A_u and A_v . It follows from Lemma 6 that $\sum_v \frac{1}{|A_v|!} \leq 1$. Since there are m different input values, $\sum_v \frac{1}{|A_v|!} \geq \sum_v \frac{1}{k!} = m/k!$. Thus $k \geq \text{fact}^{-1}(m)$. ■

Theorem 7 implies that $T_{\text{ch}} \geq \min(\text{fact}^{-1}(m), n/2)$. This matches the upper bound from Section 3 to within a small constant factor.

From Lemma 1, it follows that $T_{\text{ac}} \geq T_{\text{ch}}$. Thus, we get a lower bound for the step complexity of adopt-commit objects.

Because the requirements for conflict detectors are safety properties, we can show that the lower bound applies to randomized anonymous implementations of conflict detectors, as well.

Corollary 8 *Given any anonymous randomized implementation of an m -valued conflict detector for n processes, there is an input v such that any solo execution of $\text{check}(v)$ has step complexity at least $\min(\text{fact}^{-1}(m), n/2)$ with probability 1 against an oblivious adversary.*

Proof: Suppose not. Then, for any input v , there is some sequence of coin-flip outcomes that causes a process p_v with input v to complete a solo execution of $\text{check}(v)$ in less than $\min(\text{fact}^{-1}(m), n/2)$ steps. For each v , let E_v be the execution of the deterministic protocol obtained by fixing the coin-flips to have these outcomes. The proof of Theorem 7 constructs a combined execution E in which two processes p_u and p_v with different inputs both return **false**. Such an execution occurs with nonzero probability in the randomized algorithm, because p_u , p_v , and all of their respective clones can generate these fixed sequences of coin-flip outcomes. This violates the correctness of the implementation. ■

The corresponding bound also holds for anonymous randomized implementations of adopt-commit objects.

5 Extension to non-anonymous algorithms

In Section 5.1, we show how the lower bound of Theorem 7 for anonymous adopt-commit implementations can be extended to deterministic *non*-anonymous implementations, at the cost of greatly reducing the part of the bound that depends on n . Some of this reduction is necessary: In Section 5.2, we show that, if processes have identities, it is possible to construct a conflict detector for arbitrarily many values with $O(\log n)$ worst-case step complexity, which is substantially smaller than the $\Omega(n)$ lower bound for anonymous conflict detectors.

5.1 Lower bound for non-anonymous conflict detectors

In the anonymous lower bound, once the process in a solo execution E_v writes to a register, we can use delayed writes by clones of that process to ensure that, when E_v is interleaved with any other solo execution E_u , it will not see a value written to that register at a later step in E_u . Without anonymity, we no longer have clones to use for this purpose.

Instead, we can use different processes with the same input value, if enough of them write to the same location. However, if the number of shared registers is very large or unbounded, the processes may spread out too much for this to work. If this occurs, we adopt a different strategy and crash all processes that write to a register visible in some other execution E_u ; this is organized by **reserving** a small fraction of the registers for each input value and only allowing processes with that input value to access those registers. The vast majority of registers remain **unreserved**.

As in the anonymous case, we construct an execution E_v , for each input value v , that is organized into t rounds. Now a round may consist of many operations on many different registers. We preserve the structure of the anonymous lower bound: Each round we allow at most one more unreserved register to be accessed by processes in E_v , but only if enough processes access it simultaneously.

Enforcing these restrictions depends on the adversary's ability to crash processes that want to violate them. In fact, our construction crashes the almost all remaining processes in each round. A consequence is that it requires a very large initial supply of processes, exponential in the number of values, m , and the number of rounds, t . It also requires careful management of the number of survivors at each step.

Choosing a good partition of the registers appears to be difficult. Instead, we apply the probabilistic method and show that, for an appropriate probability distribution, there is a nonzero probability enough processes survive each round.

A complication in this approach is that a particularly perverse algorithm might exploit knowledge of the register partition to force too many processes to be crashed. To limit the algorithm's ability to determine information about the partition, even attempts to read registers reserved for processes' own input values result in crashes most of the time, if that register has not previously been accessed. Then the fact that some process crashed after attempting to access a register gives only limited information about the part of the partition to which it belongs. We make this observation rigorous using conditional probabilities. (See Lemma 10).

We now state our main lemma:

Lemma 9 *For any deterministic implementation of an m -valued conflict detector for n processes with $m \geq 150$, $t > 0$, and $n \geq 8^t m^{3t} t^{3t+1}$, there is a partition of the registers, which reserves one part for each possible input value and leaves one part unreserved, and, for each input value v , there is a t round execution E_v by processes with input value v , such that, in each round,*

1. Every process that has not crashed or been delayed to cover some register takes one step.
2. No process accesses a register reserved for an input value other than v .
3. If one or more processes read an unreserved register that was written to in an earlier round of E_v , then these reads are immediately preceded by a delayed write to that register by a process that then crashes.
4. At most one unreserved register is accessed that has not been accessed in an earlier round of E_v . If there is such a register, either all operations that access this register are reads or the first operation that accesses this register is a write and the number of write operations to this register that are delayed is at least the number of future rounds.

The following section is devoted to the proof of this lemma.

5.1.1 Proof of Lemma 9

Initially, the adversary randomly assigns registers to different input values v (representing reserved registers) or to the value $*$ (representing unreserved registers). Massive slaughter of processes at each round is used to eliminate processes that might otherwise access the wrong registers. During the construction, the adversary uses more randomness to make it difficult for the processes to identify which registers have been assigned to which values, by randomly crashing many processes that would otherwise be accessing registers reserved for their value. We show that this procedure produces a family of executions with the desired properties with nonzero probability. It follows that such a family exists.

The labels of the registers are chosen independently. Let $\ell(r) = *$ with probability $1 - \frac{1}{2tm}$. For each input value v , let $\ell(r) = v$ with probability $\frac{1}{2tm^2}$.

Having fixed a labeling, each E_v is constructed round-by-round. Say that a process is **active** if it has not crashed and is not being used to cover a register. In order to avoid crashing too many processes, the adversary budgets for at least $s_i = (8t^3m^3)^{t-i}t$ surviving active processes at the end of each round i . Letting $n = s_0m = 8^t m^{3t} t^{3t+1}$ gives the lower bound on the number of processes needed initially.

The adversary is successful if it can construct an execution that satisfies the requirements of Lemma 9 while keeping around s_i active survivors at the end of each round i .

Conversely, the goal of the algorithm is to force the adversary to crash as many processes as possible. To simplify the proof, we imagine that the algorithm is capable of coordinating the active processes perfectly, so it can deploy active processes to whatever registers it likes at each stage. We will also imagine that it learns the label of any register it successfully accesses. Note that, as long as requirement 2 of Lemma 9 holds, these labels can only be v or $*$. We call a register's label **known** if it has previously been successfully accessed.

In each round i , all active processes with pending operations on registers with known label v survive. All active processes with pending reads on registers

with known label $*$ also survive, as do all active processes with pending writes on registers with known label $*$ to which a write has occurred in a previous round of E_v .

For each register r with known label $*$, to which no writes have occurred in any previous round of E_v , if there are at least $t - i + 1$ active processes with pending writes to r , delay $t - i$ of these writes and let the remaining active processes with pending writes to r register survive. (They perform their writes before the processes that read from r .) Otherwise, crash all active processes with pending writes to r . Let s'_i denote the number of surviving active processes with pending accesses to registers with known labels.

If $s'_i \geq s_i$, then all other active processes are crashed. Otherwise, the adversary will attempt to obtain at least $s_i - s'_i$ additional survivors from among active processes with pending accesses to registers with unknown labels.

First, suppose there is some register with an unknown label to which at least $s_i - s'_i$ active processes have pending reads or to which at least $s_i - s'_i + t - i + 1$ processes have pending accesses. Let r be the register with smallest index with this property. Note that this choice of register does not depend on the register labeling. This prevents the algorithm from learning anything about the labels of registers that are not chosen. If r is reserved for v , let all the active processes with pending operations on r survive and crash all active processes with pending accesses on other registers with unknown labels. If r is reserved for a value other than v , we declare the adversary's strategy to have failed. We bound the probability that this bad event occurs below.

If r is unreserved, then all active processes with pending accesses on other registers with unknown labels are crashed. If there are at least $s_i - s'_i$ active processes with pending reads from r , then also crash all processes with pending writes to r . Otherwise, there are at least $s_i - s'_i + t - i + 1$ processes with pending accesses to r , of which at least $t - i + 1$ are writes. Delay $t - i$ of the writes and schedule the remaining writes to r before the reads from r . In either case, requirement 4 of Lemma 9 is satisfied.

If none of the previous cases apply, we crash all active processes with pending accesses on registers with unknown label other than v . For each register with unknown label v , with probability $1 - 1/t$, we crash every process that attempts to access r in this round and, with probability $1/t$, we allow all processes to successfully perform their accesses on r . If there are fewer than $s_i - s'_i$ processes that successfully access registers with unknown label v , the adversary's strategy also fails.

Observe that which branch of the adversary's strategy applies is entirely controlled by the algorithm's choices of where to place pending operations by the active processes. These choices in turn depend on the outcome of previous rounds. For a fixed algorithm, we can think of the execution E_v as a random variable that depends only on the labels of the registers and the random choices made in the last branch. We will let E_v^i denote the first i rounds of E_v . This is also a random variable.

A maximally perverse algorithm will mostly avoid having processes access registers with known labels. This reduces the analysis to two cases, depending

on whether the algorithm sends sufficiently many active processes to a single register unknown label. In both cases, success depends on the labels of the unknown registers. Because the algorithm can use information from previous rounds, the distribution of these labels conditioned on the algorithm's knowledge is not the same as the original distribution. We begin by showing that it does change by much.

Lemma 10 *Let E be any specific i -round partial execution after which register r has an unknown label. Then:*

$$\begin{aligned} \Pr[\ell(r) = * \mid E_v^i = E] &\geq 1 - \frac{1}{2tm}, \text{ and} \\ \Pr[\ell(r) = v \mid E_v^i = E] &\geq \frac{1}{8tm^2}. \end{aligned}$$

Proof: First observe that, as the labels of the registers are independent, any event in E that does not involve r has no effect on the conditional distribution of $\ell(r)$. Similarly, any event involving r whose outcome does not depend on $\ell(r)$ also has no effect; this includes attempts to access r by processes that are crashed when there are sufficiently many processes accessing registers with known labels and in the case that there are sufficiently many processes accessing the same register with with unknown label. This leaves only the last case, in which the choice of whether or not to crash a process depends on $\ell(r)$.

To formalize this intuition, let A be the event that all of the following are consistent with $E_v^i = E$: the labels of all registers other than r and, in each of the first i rounds, whether the last case of the adversary's strategy is applied and, if so, whether to crash the processes accessing registers other than r . Note that $\ell(r)$ is independent of A .

In the following, we assume $t \geq 2$; if not, then E_v^i is empty and conditioning on it has no effect.

Let C be the event that every attempt in E_v^i to access r fails, so the event $E_v^i = E$ is the intersection of the events A and C . This implies that conditioning on $E_v^i = E$ is equivalent to conditioning on both A and C . Note that $\Pr[C \mid A \cap \ell(r) = *] = 1$ and $\Pr[C \mid A \cap \ell(r) = v] = (1-1/t)^j \geq (1-1/t)^t \geq (1-1/2)^2 = 1/4$, where j is the number of rounds of E in which the last case of the adversary's strategy is applied and at least one process attempts to access register r . The last inequality holds because $(1-1/t)^t$ is increasing in t for $t \geq 1$.

Let L be the event $\ell(r) = x$, so $\Pr[L \mid A] = \Pr[L]$. Applying the definition of conditional probability, we have:

$$\Pr[L \mid C \cap A] = \frac{\Pr[C \mid L \cap A] \cdot \Pr[L \mid A]}{\Pr[C \mid A]} \geq \Pr[C \mid L \cap A] \cdot \Pr[L].$$

Thus $\Pr[\ell(r) = * \mid C \cap A] \geq \Pr[\ell(r) = *] = 1 - \frac{1}{2tm}$ and $\Pr[\ell(r) = v \mid C \cap A] \geq \Pr[\ell(r) = v]/4 = \frac{1}{8tm^2}$. ■

The first inequality immediately gives:

Corollary 11 *Suppose that, in round i , after E_v^{i-1} , the algorithm forces the first case of the adversary's strategy (in which there are sufficiently many processes accessing the same register with unknown label). Then the probability that the adversary strategy fails in round i conditioned on this event is less than $\frac{1}{2tm}$.*

Proof: The probability of failure is just the conditional probability that $\ell(r)$ is neither v nor $*$ for the chosen register r . ■

We can assume that the algorithm will avoid allowing any processes to survive by having them access registers with known labels. The algorithm can force the adversary to crash up to $t - i$ processes for each register with known label $*$, to which no process has written during E_v^{i-1} . There are at most $i - 1$ such registers, since at most one additional unreserved register is made known each round. This causes at most $(t - i)(i - 1) < t^2$ processes to crash. Thus, there are more than $s_{i-1} - t^2$ active processes that will have pending accesses to registers with unknown labels at the end of round $i - 1$.

If the algorithm forces the last case of the adversary's strategy in round i , then there are at most $s_i - t - i$ processes with pending accesses to each register r with unknown label. These processes survive only if $\ell(r) = v$ and the biased coin-flip for register r has the outcome with probability $1/t$. From Lemma 10, the probability of the first event (conditioned on E_v^{i-1}) is at least $\frac{1}{8tm^2}$, giving a conditional probability of at least $\frac{1}{8t^2m^2}$ that the processes accessing some register r survive.

Translating this into a bound on the number of survivors requires a mild variant on standard Chernoff bounds, which we state and prove below.

Lemma 12 *Let $X_i, i = 1 \dots k$ be independent 0-1 random variables each with expectation p , and let $S = \sum_{i=1}^k w_i X_i$ where, $0 \leq w_i \leq M$ for some M . Let $\mu = \mathbb{E}[S] = p \sum_{i=1}^k w_i$. Let $0 \leq \delta \leq 1$. Then*

$$\Pr[S \leq (1 - \delta)\mu] \leq \left(\frac{e^{-\delta}}{(1 - \delta)^{(1 - \delta)}} \right)^{\mu/M}. \quad (1)$$

Proof: The proof follows the standard Chernoff bound proof, augmented

with a mild appeal to convexity. Compute

$$\begin{aligned}
\mathbb{E}[e^{\alpha S}] &= \mathbb{E}\left[\exp\left(\sum_{i=1}^k \alpha w_i X_i\right)\right] \\
&= \mathbb{E}\left[\prod_{i=1}^k e^{\alpha w_i X_i}\right] \\
&= \prod_{i=1}^k \mathbb{E}[e^{\alpha w_i X_i}] \\
&= \prod_{i=1}^k ((1-p) + p e^{\alpha w_i}) \\
&= \prod_{i=1}^k (1 + p(e^{\alpha w_i} - 1)) \\
&\leq \prod_{i=1}^k \exp(p(e^{\alpha w_i} - 1)) \tag{2} \\
&= \exp\left(\sum_{i=1}^k p(e^{\alpha w_i} - 1)\right) \\
&= \exp\left(p \sum_{i=1}^k (e^{\alpha w_i} - 1)\right) \\
&\leq \exp\left(p \sum_{i=1}^k ((1 - w_i/M)(e^{\alpha \cdot 0} - 1) + (w_i/M)(e^{\alpha M} - 1))\right) \tag{3} \\
&= \exp\left(p \left(\sum_{i=1}^k w_i/M\right) (e^{\alpha M} - 1)\right) \\
&= \exp((\mu/M)(e^{\alpha M} - 1)).
\end{aligned}$$

Step (2) uses the inequality $1 + x \leq \exp(x)$, while step (3) uses the convexity of the function $\exp(\alpha x) - 1$.

When $\alpha < 0$, we have

$$\begin{aligned}
\Pr[S \leq (1 - \delta)\mu] &= \Pr[e^{\alpha S} \geq e^{\alpha(1 - \delta)\mu}] \\
&\leq \frac{\mathbb{E}[e^{\alpha S}]}{e^{\alpha(1 - \delta)\mu}} \\
&= \exp((\mu/M)(e^{\alpha M} - 1) - \alpha(1 - \delta)\mu).
\end{aligned}$$

In particular, when $\alpha = \ln(1 - \delta)/M$, we get

$$\begin{aligned} \Pr[S \leq (1 - \delta)\mu] &\leq \exp((\mu/M)((1 - \delta) - 1) - (\mu/M)(1 - \delta)\ln(1 - \delta)) \\ &= \left(\frac{e^{-\delta}}{(1 - \delta)^{(1 - \delta)}} \right)^{\mu/M}, \end{aligned}$$

as claimed. ■

We thus have:

Lemma 13 *Let $m \geq 150$ and $t > 0$. Suppose that there are at least s_{i-1} survivors at the end of E_v^{i-1} and, after E_v^{i-1} , the algorithm forces the last case of the adversary strategy in round i . Then the probability that there are fewer than s_i survivors after E_v^{i-1} , conditioned on E_v^{i-1} , is strictly less than $\frac{1}{2tm}$.*

Proof: First let us compute a lower bound on the expected number of survivors.

As argued previously, at least $s_{i-1} - t^2$ processes attempt to access registers with unknown labels in round i and the probability that each such process survives is at least $\frac{1}{8t^2m^2}$, by Lemma 10 and the fact that the processes accessing a register with label r survive with probability $1/t$.

Let S be the random variable counting the number of survivors. Then

$$\begin{aligned} \mu &= \mathbb{E}[S] \\ &\geq \frac{1}{8t^2m^2} (s_{i-1} - t^2) \\ &= \frac{8s_i t^3 m^3 - t^2}{8t^2m^2} \\ &= s_i t m - \frac{1}{8m^2} \\ &\geq s_i t m / 2 \\ &\geq 2s_i. \end{aligned}$$

The last inequality uses the fact that $tm \geq 4$.

Because at most $s_i + t - i$ process attempt to access any one register r , we can let $M = 2s_i > s_i + t - i$ and get $\mu/M \geq \frac{s_i t m / 2}{2s_i} = tm/4$. Now fix $\delta = 1/2$, and apply (1) to get

$$\begin{aligned} \Pr[S \leq s_i] &\leq \Pr \left[S \leq \frac{1}{2} s_i (tm/2) \right] \\ &\leq \Pr[S \leq (1 - \delta)\mu] \\ &\leq \left(\frac{e^{-\delta}}{(1 - \delta)^{(1 - \delta)}} \right)^{\mu/M} \\ &\leq (\sqrt{2}/e)^{tm/4}. \end{aligned}$$

Because $(\sqrt{2/e})^{tm/4}$ drops exponentially fast, it is less than $\frac{1}{2tm}$ when tm is sufficiently large. A numerical calculation shows that $tm \geq 150$ is enough. ■

To complete the proof of Lemma 9, observe that the m executions $E_1 \dots E_m$ include mt rounds between them. In each of these rounds, the probability that the adversary strategy fails is at most $\frac{1}{2tm}$, by either Corollary 11 or Lemma 13, depending on whether the algorithm forces the first or last case of the adversary's strategy. These probabilities over all mt rounds gives a total probability of failure at most $1/2$. It follows that the adversary strategy succeeds with nonzero probability, and that a set of executions as described in Lemma 9 exists.

5.1.2 Interleaving the executions

The next step is to show how these executions can be interleaved to violate the requirements of a conflict detector if t is too small.

Given an execution E_v of many copies of `check`(v) as constructed in Lemma 9, let W_v be the set of unreserved registers r that processes in E_v write to, and let X_v be the set of unreserved registers that processes in E_v read from but do not write to. Let A_v be the permutation of $W_v \cup X_v$ arranged in the order in which registers in W_v are first written and the registers in X_v are last read in E_v .

The following is analogous to Lemma 5 for the anonymous case:

Lemma 14 *For all distinct input values u and v , if each of E_u and E_v include a process that returns **false**, then there exist two registers R_i, R_j in $(W_u \cup X_u) \cap (W_v \cup X_v)$ that occur in different orders in A_u and A_v .*

Proof: Suppose otherwise. We construct an execution E that interleaves E_u and E_v and which is indistinguishable from E_u and E_v by the processes in each of these executions.

The interleaving is done round-by-round. For each register r that occurs in both A_u and A_v , we schedule the rounds in which they are first written together if $r \in W_u \cap W_v$, the rounds in which they are last read together if $r \in X_u \cap X_v$, and the rounds in which r is last read / first written together otherwise. Within each group, we schedule rounds that only read r before rounds that write r . Remaining rounds are interleaved arbitrarily. This interleaving is feasible as long as the orderings A_u and A_v are consistent.

With this interleaving, in any round where a register r in $A_u \cap A_v$ is read in by a process with input u , it observes either (a) a value written in in the same round of E_u (Case 4 of Lemma 9); (b) a value written by a write delayed from a previous round of E_u (Case 3); or (c) nothing (Case 4, if this and any previous rounds in E_u that access r perform only reads). In each case E is indistinguishable from E_u to processes with input u ; the same argument holds for v .

Any registers not in $A_u \cap A_v$ are never accessed by process in both executions. So reads of these registers in by processes with input u return only values previously written by processes with input u . Again we have that E is indistinguishable from E_u .

Suppose now that processes in both E_u and E_v return false. Then the same occurs in E , violating the specification of a conflict detector. ■

5.1.3 Proof of the lower bound

It follows that when m and n are large enough to apply Lemma 9, we can apply Lemma 6 to get $t! \geq m$, where t is the minimum length of any execution of check. When n is unbounded, this gives $t = \Omega\left(\frac{\log m}{\log \log m}\right)$ as in the anonymous case.

For bounded n , we need $8^t m^{3t} t^{3t+1} \leq n$ to make Lemma 9 work, which is equivalent to $t \log 8 + 3t \log m + (3t + 1) \log t \leq \log n$. Setting $m = \sqrt{\log n}$ and $t = \text{fact}^{-1}(m) = \Theta\left(\frac{\sqrt{\log n}}{\log \log n}\right)$ satisfies this bound for sufficiently large m , giving a combined bound of:

Theorem 15 *In any deterministic implementation of a conflict detector for n process with m input values, there is an execution in which some process takes*

$$\Omega\left(\min\left(\frac{\log m}{\log \log m}, \frac{\sqrt{\log n}}{\log \log n}\right)\right)$$

steps.

The main differences between this bound and the anonymous bound are the assumption that the algorithm is deterministic and a much stronger dependence on n . We can show that at least part of this second difference is necessary, by showing that removing anonymity allows for a more efficient conflict detector for bounded n and unbounded m .

5.2 Improved upper bounds for non-anonymous conflict detectors

For anonymous adopt-commit objects, the best bound we can get on the running time as a function of n alone is $O(n)$. This is not the case when processes have identities. In this case, we can build a conflict detector that supports arbitrarily many values by placing the processes at the leaves of a strict binary tree T with n nodes, $O(\log n)$ height, and a 2-valued conflict detector at each internal node. This is analogous to the classic tournament algorithm for mutual exclusion of Peterson and Fischer [21]. The conflict detector at a node is implemented by a pair of registers, one at each of its two children.

Pseudocode is given in Algorithm 5. The essential idea of the algorithm is that a process detects no conflict if it detects no conflict at any level on the path from its leaf to the root, where a conflict is detected at an internal node if distinct values have been written to the registers at its two children.

To show this works, we first prove a simple fact.

Lemma 16 *All values written to the register at any particular node during an execution of Algorithm 5 are the same.*

```

1 procedure check( $v$ ) for process with id  $i$ :
2 begin
3    $x \leftarrow i$ 'th leaf of  $T$ 
4   while  $x \neq T$ .root do
5      $x$ .register  $\leftarrow v$ 
6     if  $x$ .sibling.register  $\notin \{v, \perp\}$  then
7       return true
8     end
9      $x \leftarrow x$ .parent
10  end
11  return false
12 end

```

Algorithm 5: Tree-based conflict detector

Proof: By induction. Fix any execution of the algorithm. Only process i writes to the register at the i 'th leaf of T and it writes there at most once. Thus the claim is true for the leaves of T .

Now consider any internal node x and assume the claim is true for both children of x . Suppose that process p writes v to x .register and p' writes v' to x .register. A process only writes to the register at an internal node after it has written the same value to the register at one of its children. Therefore, if p and p' previously wrote to the register at the same child of x , then $v = v'$ by the induction hypothesis.

Otherwise, p wrote v to the register at node y and p' wrote v' to the register at node y' , where y and y' are the two children of node x . Suppose p wrote to y .register first. Then p' cannot read \perp from y .register, so, by the induction hypothesis, it read value v . This implies that $v' = v$; otherwise p' returns **true** before writing to x .register. ■

Theorem 17 *Algorithm 5 implements a conflict detector with $O(\log n)$ worst-case step complexity.*

Proof: If two processes have different inputs, they can't both write to the same child of the root. If they write to different children of the root, both can't read \perp from the other child of the root, so at least one returns **true**.

If all processes have the same input v , no value other than v is written to any register, so no process reads a value other than v or \perp . In this case, no process returns **true**.

The step complexity of the algorithm is proportional to the height of T , which is $O(\log n)$. ■

There is still a substantial gap between the $O(\log n)$ upper bound in Theorem 17 and the $\Omega\left(\frac{\sqrt{\log n}}{\log \log n}\right)$ lower bound in Theorem 15. We conjecture that $O(\log n)$ is closer to the correct value.

5.3 Non-anonymous randomized implementations

In the anonymous case, the lower bound on a deterministic m -valued conflict detector translates directly into a lower bound on the fastest possible solo execution of even a randomized implementation with an oblivious adversary (Corollary 8). This result depends on cloning and does not apply in the non-anonymous case. However, we can say something about the expected worst-case running time.

Theorem 18 *In any randomized implementation of a conflict detector for n processes and m input values, there is an input v and an oblivious adversary that runs n/m processes with input v , so that with probability 1, there is at least one process that takes*

$$\Omega\left(\min\left(\frac{\log m}{\log \log m}, \frac{\sqrt{\log n}}{\log \log n}\right)\right)$$

steps.

Proof: Fix t . Let $E_v(r)$ be the execution constructed in the proof of Lemma 9 when the random bits provided to the processes are fixed to the sequence r (thus making the algorithm deterministic). If the conditions of the theorem do not hold, there is a nonzero probability that $E_v(r)$ finishes in at most t steps for all v for some t that is less than the claimed lower bound; in particular, there is some fixed sequence r_0 such that all E_v violate the bound when the adversary correctly guesses that the algorithm's bits are r_0 , an event that occurs with probability $(\Pr[r = r_0])^2 > 0$. But then Theorem 7 implies that there is an interleaving of two of these executions in which processes in both return **false**, and an oblivious adversary can find this interleaving with nonzero probability. ■

This is a much weaker result than Corollary 8. In the anonymous case, we could guarantee that no process beat the lower bound; here, we can only say that some process doesn't, but it may be that almost all processes finish in $O(1)$ time and that any particular process finishes in $O(1)$ expected time. Whether or not this is actually possible remains open.

6 Consequences for consensus

Here we consider the effect of our improved bounds for adopt-commit objects on the consensus problem. In the consensus problem, n processes must agree on a value, which must be equal to some process's input. A protocol is randomized wait-free if, in addition, any process can complete it in a finite expected number of steps, regardless of the timing of the other's processes' steps or the occurrence up to $n - 1$ crash failures.

The cost of consensus depends strongly on the power of the adversary scheduler that controls timing and process failures and, to a lesser extent, on the number of possible values. For an **adaptive adversary**, which can observe the

internal states of the processes, there is a tight bound of $\Theta(n)$ on the individual step complexity of binary (two-valued) consensus [4, 6]. The high cost of consensus in this model has led to examination of models with weaker adversaries, particularly adversaries that are prevented from changing the schedule based on coin-flip values known only to one process.

One approach is to limit the adversary’s ability to observe the state of the system. A **value-oblivious adversary** [8–10] cannot observe the internal states of processes, the contents of registers, or pending operations. It bases its choice of schedule only on the history of which operations the processes have applied to which registers. The best currently known protocol in this model, due to Aumann [8], achieves consensus with $O(\log n)$ expected individual step complexity for any number of input values.

An alternative is to give extra power to the algorithm by allowing **probabilistic writes** [1, 11, 12], where a process can flip a coin and choose to execute a write operation or not based on the outcome of the coin-flip, without affecting the scheduling done by the adversary. In this model, a protocol of Aspnes [3], based on combining adopt-commit objects and a class of randomized objects called **conciliators**, gives an anonymous protocol for m -valued consensus with expected $O(\log m + \log n)$ individual step complexity, where $O(\log m)$ is the cost of the adopt-commit and $O(\log n)$ is the cost of the conciliator using implementations given in [3].

An **oblivious adversary** that must fix the schedule in advance, without seeing the actions of the processes, gives an even stronger model than both the value-oblivious and probabilistic-write models. (As observed in [3], a process in an oblivious-adversary model can simulate a probabilistic write by choosing randomly between carrying out a write and a dummy operation.) In this model, Attiya and Censor-Hillel [7] have shown that any protocol with two input values runs for at least k steps with probability c^{-k} for some constant c , a bound that translates into constant expected individual step complexity.

Our results improve the previous upper bound for the probabilistic-write model and give a non-trivial lower bound on expected individual step complexity for the oblivious-adversary model when the number of input values m is $\omega(1)$. For the probabilistic-write model, substituting our improved adopt-commit implementations for the adopt-commit object in [3] reduces the expected individual step complexity from $O(\log m + \log n)$ to $O\left(\min\left(\frac{\log m}{\log \log m}, n\right) + \log n\right)$ in the anonymous case and $O(\log n)$ in the non-anonymous case. For the oblivious-adversary model, our lower bound on anonymous adopt-commit objects gives an immediate $\Omega\left(\min\left(\frac{\log m}{\log \log m}, n\right)\right)$ lower bound with probability 1 on the worst-case individual step complexity of anonymous m -valued consensus implementations, because consensus objects satisfy the specification of adopt-commit objects. This is the first lower bound for consensus for which the number of values m is significant.

7 Conclusions

We have shown how to reduce adopt-commit objects to conflict detectors, which are much simpler, and used this reduction to get tight bounds on the individual step complexity of anonymous m -valued adopt-commit objects. These bounds also translate into improved bounds on anonymous m -valued consensus. We have also shown bounds for the deterministic non-anonymous case, which give an exponential separation between the anonymous and non-anonymous cases as a function of n .

The natural question is whether it is possible to get tight bounds for non-anonymous implementations, and whether randomization could allow further improvements here.

References

- [1] Karl Abrahamson. On achieving consensus using a shared memory. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 291–302, 1988.
- [2] Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. Of choices, failures and asynchrony: The many faces of set agreement. In Yingfei Dong, Ding-Zhu Du, and Oscar H. Ibarra, editors, *ISAAC*, volume 5878 of *Lecture Notes in Computer Science*, pages 943–953. Springer, 2009.
- [3] James Aspnes. A modular approach to shared-memory consensus, with applications to the probabilistic-write model. In *Proceedings of the Twenty-Ninth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 460–467, July 2010.
- [4] James Aspnes and Keren Censor. Approximate shared-memory counting despite a strong adversary. In Claire Mathieu, editor, *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*, pages 441–450. SIAM, 2009.
- [5] James Aspnes, Faith Ellen Fich, and Eric Ruppert. Relationships between broadcast and shared memory in reliable anonymous distributed systems. *Distributed Computing*, 18(3):209–219, February 2006.
- [6] Hagit Attiya and Keren Censor. Tight bounds for asynchronous randomized consensus. *J. ACM*, 55(5):20, 2008.
- [7] Hagit Attiya and Keren Censor-Hillel. Lower bounds for randomized consensus under a weak adversary. *SIAM J. Comput.*, 39(8):3885–3904, 2010.
- [8] Yonatan Aumann. Efficient asynchronous consensus with the weak adversary scheduler. In *PODC '97: Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 209–218, New York, NY, USA, 1997. ACM.

- [9] Yonatan Aumann and Michael A. Bender. Efficient low-contention asynchronous consensus with the value-oblivious adversary scheduler. *Distributed Computing*, 17(3):191–207, 2005.
- [10] Tushar Deepak Chandra. Polylog randomized wait-free consensus. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 166–175, Philadelphia, Pennsylvania, USA, 23–26 May 1996.
- [11] Ling Cheung. Randomized wait-free consensus using an atomicity assumption. In *Principles of Distributed Systems, 9th International Conference, OPODIS 2005, Pisa, Italy, December 12-14, 2005, Revised Selected Papers*, volume 3974 of *Lecture Notes in Computer Science*, pages 47–60. Springer, 2006.
- [12] Benny Chor, Amos Israeli, and Ming Li. Wait-free consensus using asynchronous hardware. *SIAM J. Comput.*, 23(4):701–712, 1994.
- [13] Faith Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45:843–862, September 1998.
- [14] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.
- [15] Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract). In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, pages 143–152, 1998.
- [16] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
- [17] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
- [18] Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, pages 163–183, 1987.
- [19] D. A. Lubell. A short proof of Sperner’s lemma. *Journal of Combinatorial Theory A*, 1(2):402, 1966.
- [20] Achour Mostefaoui, Sergio Rajsbaum, Michel Raynal, and Corentin Travers. The combined power of conditions and information on failures to solve asynchronous set agreement. *SIAM Journal on Computing*, 38(4):1574–1601, 2008.

- [21] Gary L. Peterson and Michael J. Fischer. Economical solutions for the critical section problem in a distributed system (extended abstract). In John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison, editors, *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, pages 91–97. ACM, 1977.