Tight Bounds for Anonymous Adopt-Commit Objects

James Aspnes [*]
Department of Computer Science
Yale University
New Haven, Connecticut, USA
aspnes@cs.yale.edu

Faith Ellen [†]
Department of Computer Science
University of Toronto
Toronto, Ontario, Canada
faith@cs.toronto.edu

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 **weak conflict-detectors**.

It follows that the same lower bound holds on 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 also be used to slightly improve the cost of randomized consensus in the probabilistic-write model.

Categories and Subject Descriptors

F.1.2 [**Modes of Computation**]: Parallelism and concurrency

General Terms

Algorithms, Theory

Keywords

distributed computing, shared memory, anonymity, adopt-commit objects, conflict detectors, consensus

1. INTRODUCTION

An **adopt-commit object** [2] or **ratifier** [3] is a one-shot shared-memory object that represents the **adopt-commit protocols** of [14] and can be used to implement round-based protocols for set-agreement and consensus. An m-valued adopt-commit object supports a single operation, `adopt-Commit` (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 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. In addition, as observed in [3], lower bounds on adopt-commit objects also yield immediate lower bounds on consensus.

The requirements for an adopt-commit object are:

1. **Validity.** Every operation's output equals some operation's input.

2. **Termination.** Every operation finishes its operation in a finite number of steps with probability 1, where the probability is taken over the coin tosses performed by the algorithm.

3. **Coherence.**[1] If some operation returns (commit, v), every operation returns either (adopt, v) or (commit, v).

4. **Convergence.** If all inputs are v, all operations return (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 processes obtain the same output) 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. 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 [14] 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

[*]Supported in part by NSF grant CCF-0916389.

[†]Supported in part by the Natural Science and Engineering Research Council of Canada.

[1]The 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].

they are scheduled. A number of advantages of anonymity are 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 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.

2. CONFLICT DETECTORS

The implementation of an adopt-commit object in [3] relies on a quorum-based conflict detection mechanism, where each process with value v writes to a set of registers W_v and detects conflicting values by reading a set of registers R_v, with the property that $W_v \cap R_{v'} \neq \emptyset$ when $v \neq v'$. It is shown there that the smallest possible size for these quorums is $\Theta(\log m)$, using tools from extremal combinatorics. The quorum-based mechanism generalizes a similar mechanism in [14], where a process detects conflicting values by performing a collect over single-writer registers, which requires individual step complexity linear in the number of processes.

If we set aside the quorum structure, we can define an abstract conflict-detector object as a generalization of these mechanisms. We begin with a linearizable version that can be used as a drop-in replacement for existing conflict detection mechansims. Then we further reduce it to a weaker (and, thus, easier to implement) version that does not satisfy linearizability.

Formally, an **m-valued strong conflict-detector** supports two operations, `write(`v`)`, for inputs v from a set of m values, and `read()`, where `read()` returns **true** (conflict) if two or more different values have previously been written and returns **false** (no conflict), otherwise. These operations must appear to be atomic to the user of the strong conflict-detector. Specifically, we require that any strong conflict-detector implementation be **linearizable** [15], meaning that, for any concurrent execution of strong conflict-detector operations, we can construct 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.

Linearizability is sometimes difficult to prove. To make things simpler, we show that strong conflict-detectors can be built from even weaker objects, which we call **weak conflict-detectors**. An m-valued weak conflict-detector supports only 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 argument, they all return **false**. For weak conflict-detectors, we do not require linearizability: it is fine for one `check` operation to return **true** while subsequent `check` operations return false.

2.1 Equivalence of adopt-commit objects, strong conflict-detectors, and weak conflict-detectors

We show that the individual step complexities of adopt-commit objects, strong conflict-detectors, and weak conflict-detectors differ by small additive constants. Because our reductions are anonymous, this also holds for anonymous implementations. We use $T_{\texttt{adoptCommit}}$, $T_{\texttt{write}}$, $T_{\texttt{read}}$, and $T_{\texttt{check}}$ to denote the worst case step complexities of the `adoptCommit`, `write`, `read`, and `check` operations.

We give implementations of a weak conflict-detector from an adopt-commit object, a strong conflict-detector from a weak conflict-detector, and finally an adopt-commit object from a strong conflict-detector. We begin by showing how to implement a weak conflict-detector from an adopt-commit object. This is is the simplest case, since neither of these objects is required to satisfy linearizability. The code is presented in Figure 1.

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

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

LEMMA 1. *Algorithm 1 implements a weak conflict-detector with* $T_{\texttt{check}} = T_{\texttt{adoptCommit}}$.

PROOF. If all `check` operations have the same input v, then, they all call $r.$`adoptCommit`(v), which, by the convergence property, all return (commit, v). In this case, all the `check(`v`)` operations return **false**. If there are two operations, `check(`v`)` and `check(`v'`)`, with $v \neq v'$, then, they call $r.$`adoptCommit`(v) and $r.$`adoptCommit`(v'), respectively. By coherence, it is not possible for (commit, v) to be the result of $r.$`adoptCommit`(v) and for (commit, v') to be the result of $r.$`adoptCommit`(v') in the same execution. It follows that **true** is returned by at least one of the two `check` operations. Thus, Algorithm 1 implements a weak conflict-detector.

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

To extend a weak conflict-detector to a strong conflict-detector, we add a one-bit register, conflict, which is set to

true whenever a `write` operation detects a conflict. The code for `read` and `write` is presented in Algorithm 2. To carry out a `read` operation, a process simply reads the conflict bit and returns its value. We show, in Lemma 2, that this does, in fact, give a strong (i.e., linearizable) conflict-detector.

```
shared data:
    weak conflict-detector $d$;
    1-bit atomic register conflict, initially false.
procedure write($v$)
begin
    if $d$.check($v$) then
        conflict ← true
    end
end
procedure read()
begin
    return conflict
end
```

Algorithm 2: A strong conflict-detector using a weak conflict-detector.

LEMMA 2. *Algorithm 2 implements a strong conflict-detector with* $T_{\texttt{write}} \leq T_{\texttt{check}} + 1$ *and* $T_{\texttt{read}} = 1$.

PROOF. The running time is immediate from the code.

To show that Algorithm 2 implements a strong conflict-detector, we give an explicit linearization of the `read` and `write` operations in any execution. The linearization point, t_r, of a `read` operation, r, is the time at which it reads register conflict. Let τ be the first time during the execution at which some process sets conflict to **true**, or $+\infty$, if there is no such time. For each `write` operation w, let s_w be the time at which the `write` operation starts. The linearization point, t_w, of operation w is defined to be $\max(s_w, \tau)$, if w sets conflict to **true**, and s_w, otherwise.

We show that these assigned times (with ties broken arbitrarily) gives a correct linearization. A `write` operation that sets conflict to **true** is linearized when it starts or at the first time in the execution that some process sets conflict to **true**, whichever is later. Note that this occurs at or before the end of the operation. All other operations are linearized when they start. This linearization order is consistent with the observable execution order.

Recall that τ is the time at which conflict is set to **true**. Any read that is linearized before τ reads **false** from conflict. Similarly, any read that is linearized after τ reads **true** from conflict. We satisfy the specification of the strong conflict-detector if (a) at most one distinct value appears as an argument to any `write` operation linearized strictly before τ and (b) if $\tau \neq +\infty$, then at least two different values appear as arguments to `write` operations linearized at or before time τ.

To be linearized before τ, a `write`(v) operation must not set conflict to **true** and, hence, its call to d.`check`(v) must return **false**. But the specification of a weak conflict-detector implies that all d.`check` operations which return **false** must have the same input value. It follows that all `write` operations linearized before τ have the same input value. This proves (a).

If $\tau \neq +\infty$, then some `write` operation, w, sets conflict to **true** at time τ and $t_w = \max(s_w, \tau) = \tau$. This `write` operation previously completed a call to d.`check` that returned **true**. If all calls to d.`check` that started before τ have the same input value, then we can truncate the execution at time τ and allow all `check` operations to run to completion. This results in an execution of the weak conflict-detector d in which all calls to d.`check` have the same input value, but some call to d.`check` returns **true**, which violates the specification of a weak conflict-detector. Hence, there are two calls to d.`check` with different input values that started before τ. Thus, there must be two corresponding `write` operations w and w' with different input values that also started before τ. These are assigned linearization points $\max(s_w, \tau) = \max(s_{w'}, \tau) = \tau$. This proves (b). □

Finally, Algorithm 3 completes the cycle by showing how to turn an anonymous strong conflict-detector into an adopt-commit object, with the addition of an extra register for holding proposed values. The mechanism is essentially the same as in the adopt-commit implementation given in [3], with a generic strong conflict-detector taking the place of the quorum-based mechanism used there.

```
shared data:
    register proposal, initially ⊥;
    strong conflict-detector $c$.
procedure adoptCommit($v$)
begin
    c.write($v$)
    $u$ ← proposal
    if $u \neq \bot$ then
        $v$ ← $u$
    else
        proposal ← $v$
    end
    if $c$.read() = true then
        return (adopt, $v$)
    else
        return (commit, $v$)
    end
end
```

Algorithm 3: An adopt-commit object using a strong conflict-detector.

LEMMA 3. *Algorithm 3 implements an adopt-commit object with* $T_{\texttt{adoptCommit}} \leq T_{\texttt{write}} + T_{\texttt{read}} + 2$.

PROOF. The proof of coherence relies on the following key observation: If an `adoptCommit` operation returns (commit, v), then proposal was set to v before any `adoptCommit`(v') operation with $v' \neq v$ finished its call to c.`write`. It follows that all `adoptCommit`(v') operations with $v' \neq v$ read v from proposal and return (adopt, v). The other properties of the adopt-commit object are easily verified. □

Since the specification of a weak conflict-detector is simpler than those of strong conflict-detectors or adopt-commit objects, it will be easiest to obtain bounds on their complexity by concentrating on weak conflict-detectors.

3. UPPER BOUNDS ON ANONYMOUS WEAK CONFLICT-DETECTORS

In this section, we give two complementary implementations of anonymous m-valued weak 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 weak conflict-detector that runs in $O\left(\min\left(\frac{\log m}{\log\log m}, n\right)\right)$ steps, which we show to be optimal in Section 4.

3.1 Permutation-based weak conflict-detector

Algorithm 4 implements an anonymous, deterministic weak conflict-detector for $m \le k!$ values using at most $2k$ operations for `check(`v`)`. As a function of m, this gives a worst-case individual step complexity of $2\,\mathrm{fact}^{-1}(m) = O\left(\frac{\log m}{\log\log m}\right)$, where $\mathrm{fact}(k) = k!$ is the factorial function.

```
   shared data:
     registers $R[1..k]$, initially $\perp$.
1  procedure 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 4: Permutation-based weak conflict-detector for m values.

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 4 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 $\pi_v : \{1, \ldots, k\} \to \{1, \ldots, k\}$. Then, for any two different input values, there exist two registers which function as in the natural two-value algorithm.

LEMMA 4. *Algorithm 4 implements a weak conflict-detector.*

PROOF. If all calls to `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 $R[1], \ldots, R[k]$ and the values u and u' will both be written to all of the registers. Let $j, j' \in \{1, \ldots, k\}$ be two indexes such that j occurs before j' in π_u, but j' occurs before j in $\pi_{u'}$. If 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.

Therefore u' is written to $R[j']$ before u is written to $R[j]$. But, then, no process writes u to $R[j']$, which is also a contradiction. □

3.2 Collect-based weak conflict-detector

Algorithm 5 is another implementation of a weak 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 `check(`v`)` operation in Algorithm 5 is $3n+1$.

```
   shared data:
     registers $R[1..n]$, initially $\perp$;
     1-bit atomic register done, initially false.
1  procedure 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 5: A collect-based weak conflict-detector for n processes.

The essential idea is that once some process finishes the first loop in `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 5. *Algorithm 5 implements a weak conflict-detector.*

4. LOWER BOUND ON ANONYMOUS WEAK CONFLICT-DETECTORS

In this section, we show that any m-valued weak 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 weak conflict-detector. For each input value v, we consider the solo execution E_v in which a process executes `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 6. *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 `check(u)` and indistinguishable from E_v to some other process p_v performing `check(v)`. In this execution, both p_u and p_v return **false**, violating the specification of a weak 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' appear in the same order as in E_u. Hence $E'|p_u = E_u$. 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 $E'|p_v = E_v$.

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 lock-step 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 of Sperner's Lemma [16].

Lemma 7. *Let $\{A_1, \ldots, A_m\}$ 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 $A_1, \ldots, A_m$. 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 $\mathrm{E}[X_i] = \frac{1}{|A_i|!}$. Hence $\sum_{i=1}^{m} \frac{1}{|A_i|!} = \sum_{i=1}^{m} \mathrm{E}[X_i] = \mathrm{E}[X] \leq 1$. □

Theorem 8. *The worst-case solo step complexity of any anonymous deterministic implementation of an m-valued weak conflict detector for n processes is at least $\min(\mathrm{fact}^{-1}(m), n/2)$, where $\mathrm{fact}(\ell) = \ell!$ is the factorial function.*

Proof. Fix any anonymous deterministic implementation of an m-valued weak 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 6, there are two registers that occur in different orders in A_u and A_v. It follows from Lemma 7 that $\sum_v \frac{1}{|A_v|!} \leq 1$. Since there m different input values, $\sum_v \frac{1}{|A_v|!} \geq \sum_v \frac{1}{k!} = m/k!$. Thus $k \geq \mathrm{fact}^{-1}(m)$. □

Theorem 8 implies that $T_{\texttt{check}} \geq \min(\mathrm{fact}^{-1}(m), n/2)$. This matches the upper bound from Section 3 to within a small constant factor.

From Lemma 1, it follows that $T_{\texttt{adoptCommit}} \geq T_{\texttt{check}}$ and, from Lemma 3, it follows that $T_{\texttt{write}} + T_{\texttt{read}} \geq T_{\texttt{adoptCommit}} - 2$. Thus, we get lower bounds for the individual step complexities of adopt-commit objects and strong conflict-detectors.

Lemma 2 says that Algorithm 2 implements a strong conflict-detector with $T_{\texttt{write}} \leq T_{\texttt{check}} + 1$ and $T_{\texttt{read}} = 1$, so $T_{\texttt{write}} + T_{\texttt{read}} \leq T_{\texttt{adoptCommit}} + 2$. It is also possible to construct a strong conflict-detector with $T_{\texttt{write}} = 1$ using one Boolean register R_v for each input value v. The idea is to have `write(`v`)` set R_v to **true** and have `read()` return **true** if at least two of these m registers are **true**. Note that, for this algorithm, $T_{\texttt{write}} + T_{\texttt{read}} = m + 1$, which is much larger than the lower bound. It is still open whether an algorithm with constant $T_{\texttt{write}}$ and smaller $T_{\texttt{read}}$ is possible.

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

Corollary 9. *Given any anonymous randomized implementation of an m-valued weak conflict detector for n processes, there is an input v such that any solo execution of* `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 `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 8 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. $\square$

The corresponding bounds also hold for anonymous randomized implementations of adopt-commit objects and strong conflict-detectors.

5. 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 implementation for the adopt-commit object in [3] reduces the expected individual step complexity from $O(\log m + \log n)$ to $O\left(\min\left(\frac{\log m}{\log \log m}, n\right) + \log n\right)$. 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.

6. CONCLUSIONS

We have shown how to reduce adopt-commit objects to a much simpler class of weak conflict-detectors, 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. The natural question is what happens when the assumption of anonymity is removed. We conjecture that for unboundedly many processes, Ramsey-theoretic techniques may be used to show that similar bounds hold. However, the complexity of non-anonymous adopt-commit objects is still unknown.

7. 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] 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.

[15] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[16] D. A. Lubell. A short proof of Sperner's lemma. *Journal of Combinatorial Theory A*, 1(2):402, 1966.