A modular approach to shared-memory consensus, with applications to the probabilistic-write model

James Aspnes*
Yale University

May 14, 2011

Abstract

We show that consensus can be solved by an alternating sequence of **adopt-commit objects** [2, 25], which detect agreement, and **conciliators**, which ensure agreement with some probability. We observe that most known randomized consensus algorithms have this structure.

We give a deterministic implementation of an m-valued adopt-commit object for an unbounded number of processes that uses $\lg m + \Theta(\log \log m)$ space and individual work. We also give a randomized conciliator for any number of values in the probabilistic-write model with n processes that guarantees agreement with constant probability while using one multi-writer register, $O(\log n)$ expected individual work, and $\Theta(n)$ expected total work. Combining these objects gives a consensus protocol for the probabilistic-write model that uses $O(\log m + \log n)$ individual work and $O(n \log m)$ total work. No previous protocol in this model uses sublinear individual work or linear total work for constant m.

1 Introduction

The consensus problem [29] is to devise a protocol so that n processes, each with a private input, can agree on a single common output that is equal to some process's input. For asynchronous deterministic processes, consensus is known to be impossible with a single crash failure in either a message-passing [24] or shared-memory [28] model. These impossibility results can be overcome using randomization [18], and randomized consensus protocols can even tolerate up to $n - 1$ crash failures in a shared-memory model [1]. There is a substantial literature on randomized consensus protocols that tolerate crash failures; see [7] for a survey.

In this paper, we concentrate on **wait-free** [26] consensus protocols in various shared-memory models with only atomic registers as their base objects (see Section 2). Our goal is to find efficient algorithms, that minimize both the total number of register operations carried out by all processes (the **total work** T_{total}) and the maximum number of register operations carried out by any single process (the **individual work** $T_{\text{individual}}$).

An important factor in determining the cost of consensus is the strength of the adversary scheduler that controls the interleaving of operations, where the strength of the adversary is a consequence of what information the adversary is allowed to use when choosing each step. For an adaptive adversary, which can observe the entire state of the system when choosing which process

*Supported in part by NSF grant CCF-0916389.

moves next, optimal algorithms are known for both total expected work [13] and individual expected work [10]. But for weaker adversaries, finding tight bounds on expected work has remained an open problem.

Recently, Attiya and Censor [14] have shown that any randomized consensus protocol for 2-valued consensus tolerating f failures must fail to terminate after $k(n-f)$ total steps with probability at least $\frac{1}{c^k}$. This bound is very general: it applies even with a very weak adversary that cannot observe the protocol's execution, and works even in a model that allows global coins visible to all processes as well as $O(1)$-cost snapshot operations. They also show that it is tight for models with correlated local coins, using as a matching upper bound the BINARY-CONSENSUS protocol of [15]. Unfortunately, it gives little insight into the *expected* cost of consensus; the geometric series converges to $\Omega(n)$, a trivial lower bound on total work.

The best currently known upper bounds on expected work for weak adversaries in a asynchronous shared-memory model without correlated coins are $O(\log n)$ expected individual work (and $O(n \log n)$ expected total work) assuming a **value-oblivious adversary** that cannot observe internal states of processes or the contents of registers (but can see where processes choose to read and write) [15]; and $O(n \log \log n)$ expected total work (with the same bound on individual work) in the **probabilistic-write model**, where a process can flip a coin and choose whether or not to execute a write operation based on its outcome [22]. (We argue in Section 2 that this model is a special case of assuming a **location-oblivious adversary**, one that cannot distinguish between operations on different registers.)

The present work improves on previous upper bounds for weak-adversary randomized consensus by giving a new family of consensus protocols for the probabilistic-write model. For 2-valued consensus, we obtain $O(\log n)$ expected individual work and $O(n)$ expected total work with a single algorithm. This is the first weak-adversary consensus protocol with optimal total work, and demonstrates that the Attiya-Censor lower bound is asymptotically tight for the probabilistic-write model. For m-valued consensus, the individual work remains the same, but the total work rises to $O(n \log m)$; there may still be room for improvement in both this bound and the individual-work bound in both the binary and m-valued cases.

Our algorithms are structured as classic Las Vegas style randomized algorithms [17]. We explicitly separate the problems of producing agreement and detecting it into abstract **conciliator** objects, which produce agreement with some probability, and **adopt-commit** objects, which cause the processes to decide only once agreement has been reached. This modular approach allows these tasks to be optimized separately, with the expected cost of a consensus object built in this fashion proportional to the sum of the cost of a conciliator and adopt-commit object. Furthermore, since a consensus object satisfies the specifications for both classes of objects, a lower bound on either conciliators or adopt-commit objects will translate directly into a lower bound on consensus. This may help in determining the limits to weak-adversary consensus protocols.

Formal definitions of these objects, together with methods for composing them, are given in Section 3. Generic algorithms for consensus (that do not specify the details of their component conciliators and adopt-commit objects) are given in Section 4.

In Section 5, we give an implementation of a conciliator for arbitrarily many values that produces agreement with constant probability at the cost of $O(\log n)$ individual work in the worst case and $O(n)$ expected total work. (This dominates the cost of the adopt-commit object for binary consensus, hence our binary-consensus bound.) The implementation is very simple, and uses only a single multi-writer register large enough to hold one input value or a null value $\perp$. We also show

that **weak shared coins** as defined in [11] can be turned into conciliators. While this does not give any new results, it shows how the conciliator+adopt-commit framework can be used to model previous shared-memory consensus protocols.

In Section 6, we give a deterministic implementation of an m-valued adopt-commit object that uses $\lceil \lg m \rceil + O(\log \log m)$ registers and has the same individual work cost. For m-valued consensus, the total work cost of the adopt-commit object becomes dominant. Though we have not fully resolved the cost of consensus in weak-adversary models, we believe that our decomposition of consensus into separate conciliation and adopt-commit steps is natural, and that the limitations of our current implementations hint at where further improvements would be needed to obtain better bounds.

2 Model

We use the standard model of asynchronous shared memory. There are n processes that communicate by reading and writing atomic registers, with the value returned by each read equal to the last value written. Asynchrony is modeled by interleaving. Each process that has not halted has exactly one **pending operation** in each state; an **execution**, which consists of a sequence of operations and their return values, is constructed by repeatedly applying pending operations. A **partial execution** is one in which not all processes have halted. The choice of which pending operation occurs in each state is determined by an **adversary scheduler**, a function from partial executions to process ids. We consider only **wait-free** protocols, where there are no fairness conditions on executions.

Processes are randomized: they have access to **local coins** that are not predictable by the adversary but also not visible to other processes. Formally, we can think of the local coins as probabilistic read-only objects private to each process, so that executing a coin-flip is an operation.

The **total work** or **total step complexity** of an execution is the total number of operations it contains. The **individual work** or **individual step complexity** is the maximum number of operations carried out by any single process. Local computation (including local coin-flip operations) is excluded from both measures.

2.1 Strong and weak adversaries

The strength of the adversary has a large effect on the cost of randomized consensus. An **adaptive** or **strong** adversary has no restrictions on its choice of which process carries out the next operation. A **weak adversary** is any adversary that is not strong. Typically, this means that the adversary's knowledge is limited in some way; this can be modeled by an equivalence relation on partial executions, where the adversary is required to choose the same process after equivalent partial executions.

Some examples of weak adversaries:

- **Oblivious adversary**. An oblivious adversary has no knowledge of the execution and schedules processes in a fixed order. Here, two executions are equivalent if they have the same length.
- **Value-oblivious adversary**. A value-oblivious adversary cannot observe internal states of processes, the contents of registers, or the values in pending write operations, but it can

distinguish between pending operations of different types (e.g., read vs. write) or to different locations. Value-oblivious adversaries of various kinds are used by [15, 16, 21]. The useful property of value-oblivious adversaries is that they allow the outcome of a local coin-flip to be stockpiled in a register as a pre-flipped global coin that is visible to all processes but still not predictable by the adversary until some process uses its value.

- **Location-oblivious adversary**. A location-oblivious adversary cannot observe internal states of processes, *can* observe the contents of memory and the values of pending write operations, but cannot distinguish between pending write operations to different locations. This allows for **probabilistic writes** as defined in the "strong model" of [1] to represent the assumptions of the Chor-Israeli-Li protocol [23] and as used more recently by Cheung [22]. These are write operations that take effect only with some probability, where the adversary cannot choose whether to allow the write operation based on the outcome of the coin-flip. In a location-oblivious model, they can be implemented by having a process randomly choose between writing to the desired location or to some dummy location (corresponding to an omitted write).

For some of our results, we assume probabilistic writes, which can be implemented using a location-oblivious adversary as described above.

The practical justification for weak adversaries is that realistic implementations of shared memory are not likely to distinguish between operations with similar properties. This is a very natural assumption for value-oblivious adversaries. For location-oblivious adversaries, it may be less natural, but it is still plausible if we assume that the relevant memory locations are all stored on the same page, and that the main source of timing uncertainty lies in a page-based memory management system that treats all locations on the same page as equivalent. The same considerations apply to probabilistic writes, if we assume they are implemented as a choice between writing a real location or a dummy location with the same characteristics.

Other restrictions on the adversary have been used in the consensus literature. These include requiring that the adversary implement a quantum or priority-based scheduling algorithm [3,4,30] or that the adversary include some random jitter in its scheduling decisions [6]. We discuss implications of some of these assumptions for our framework in Section 4.2.

3 Decomposing consensus

Here we show how a consensus protocol can be decomposed into a collection of conciliators and adopt-commit objects. This provides an alternative to previous general frameworks for implementing consensus, such as the round-based weak-shared-coin framework of Aspnes and Herlihy [11].

Recall that a **randomized consensus protocol** must satisfy three properties:

- **Validity**. Every output equals some process's input.
- **Termination**. Every process terminates with probability 1.
- **Agreement**. No two outputs are different.

A **consensus object** is a shared-memory object with a single **consensus** operation, such that if each process executes this operation exactly once with its input, the resulting outputs satisfy the requirements for randomized consensus.

We are going to replace consensus objects by sequences of objects satisfying weaker conditions. Like consensus objects, these weaker objects will be **one-shot** objects—which means that each process executes at most one operation on the object—and we will expect inputs to be from the set of possible decision values Σ. But the outputs will be annotated by a **decision bit**, chosen from the set $\{\mathsf{commit}, \mathsf{adopt}\}$, that indicates whether the protocol should terminate immediately or continue to the next object in the sequence: an output of (commit, v) means to terminate with decision value v immediately, while an output of (adopt, v) means to continue to the next object, using v as input. We call an object annotated with a decision bit in this way a **deciding object**. The adopt and commit values for the decision bit are used for compatibility with the definition of adopt-commit objects given by [2].

These objects will generally satisfy only **Las Vegas** requirements [17], where an object is not required to guarantee agreement unless at least one of its outputs is marked with commit. We refer to this weaker agreement requirement as **coherence**:[1]

- **Coherence**. If any process outputs (commit, v), then no process outputs (d, v') for $v' \neq v$.

A **weak consensus object** is a deciding object that satisfies validity, termination, and coherence. Note that weak consensus objects may be very weak indeed: an object that simply copies its input to its output with decision bit adopt satisfies all three properties. But we will use weak consensus objects as a basis for building stronger objects.

3.1 Conciliators

A **conciliator** is a weak consensus object that guarantees that its outputs agree with constant probability, but that does not detect when or if agreement occurs. Formally, it satisfies the additional requirement of **probabilistic agreement**:

- **Probabilistic agreement.** There is a fixed **agreement probability** $\delta > 0$ such that, for any adversary strategy, the probability that all return values are equal is at least δ.

A conciliator object may correspond to a weak shared coin (with machinery added to ensure validity) or to the first-mover-wins technique of Chor-Israel-Li-style protocols [22, 23]. We give examples of both types of constructions in Section 5. Because conciliators are not expected to detect agreement, our constructions will satisfy coherence vacuously, by always returning adopt.

3.2 Adopt-commit objects

Actual decisions are produced by **adopt-commit objects**, objects that detect agreement. An adopt-commit object is a weak consensus object that satisfies the additional requirement of **convergence**:[2]

- **Convergence**. If all inputs are equal to v, all outputs are (commit, v).

[1] The presentation of adopt-commit objects in [2] uses the term *agreement* for this property; we will reserve *agreement* for the stronger, unconditional agreement property of full consensus objects.

[2] The terminology here follows [2]. The term used in [8] was the less evocative *acceptance*.

Adopt-commit objects [2] are a representation, as shared-memory objects, of the adopt-commit protocols of Gafni [25]. The same objects, with minor cosmetic differences, were defined under the name of *ratifiers* in [8]. The adopt-commit objects defined here largely follow the definition and terminology in [2], with the exception that we use the term *coherence* (from [8]) for what is called *agreement* in [2].

Because adopt-commit objects are not required to produce agreement, they can be implemented deterministically with low space and work complexity. In Section 6, we give an implementation of an m-valued adopt-commit object that uses $O(\log m)$ multi-writer registers and $O(\log m)$ individual work. For binary consensus, this reduces to 3 registers and at most 4 operations per process.

3.3 Composing objects

Our consensus protocols will consist of an alternating sequence of adopt-commit objects and conciliators. To define this formally, it helps to have a notion of composing deciding objects.

Recall that a deciding object is one that annotates its output with either adopt or commit. If X and Y are deciding objects then their composition $(X;Y)$ is a deciding object whose operation $\mathrm{op}_{(X;Y)}(x)$ is given by the code in Procedure `Composition`.

```
1 (d, v) ← op_X(x)
2 if d = commit then
3     return (commit, v)
4 else
5     return op_Y(v)
6 end
```

Procedure `Composition(`X, Y, x`)`

Informally, we perform op_X first, and feed the result to op_Y only if op_X does not commit to a value on its own. We can also think of this rule as implementing an exception mechanism where a commit by X immediately terminates the composite object without executing Y.

Note that in $(X;Y)$, X comes first. This is the reverse of the usual rule for function composition.

It is easy to see that composition is associative: $((X;Y);Z)$ has exactly the same behavior as $(X;(Y;Z))$ for all objects X, Y, Z. We will generally omit the parentheses and write $(X;Y;Z)$ in this case. Similarly, we can define compositions $(X_1;X_2;\ldots;X_k)$ for arbitrarily many objects, and even define infinite compositions $(X_1;X_2;\ldots)$ in the obvious way.

If P is some predicate on objects, we say that composition **preserves** P if $P(X)$ and $P(Y)$ implies $P(X;Y)$. This naturally extends by induction to longer finite compositions.

3.3.1 Composing weak consensus objects

The property of being a weak consensus object is preserved by composition.

Lemma 1 *Validity is preserved by composition.*

Proof: Suppose that X and Y satisfy validity. Then any return value of $(X;Y)$ is either (a) a return value of X, and thus equal to an input to $(X;Y)$; or (b) a return value of Y, and thus

equal to an input to Y, which is in turn an output of X and thus equal to an input of $(X;Y)$. ∎

The converse does not hold. It may be that the first object always scrambles its inputs (but does not commit) while the second unscrambles them.

Lemma 2 *Termination is preserved by composition.*

Proof: Immediate. ∎

A partial converse holds for termination: if $(X;Y)$ terminates with probability 1, so does X.

Lemma 3 *If X satisfies coherence, and Y satisfies both validity and coherence, then $(X;Y)$ satisfies coherence.*

Proof: We consider two cases, depending on whether any process skips Y:

1. X outputs (commit, v) for some process. Then coherence for X implies all processes obtain $(-, v)$ from X, and any other process either obtains the same return value directly from X or obtains the same return value from Y because of the validity of Y.
2. X does not output $(\mathsf{commit}, -)$ for any process. Then all processes execute op_Y and coherence follows from coherence of Y.

∎

Theorem 4 *The property of being a weak consensus object is preserved by composition.*

Proof: Apply Lemmas 1, 2, and 3. ∎

4 Recomposing consensus

We give three constructions of consensus protocols. The first uses an infinite sequence of adopt-commit objects and conciliators (but terminates after using only a constant number on average); the second truncates the infinite sequence by falling back to some fixed-space consensus protocol with low probability; and the third omits the conciliators and relies on scheduling restrictions to terminate.

4.1 Consensus with conciliators and adopt-commit objects

4.1.1 Unbounded construction

Let A_i be a separate adopt-commit object for each $i \geq -1$ and C_i a conciliator object with agreement probability δ for each $i \geq 1$. Construct the composite object

$$U = A_{-1}; A_0; C_1; A_1; C_2; A_2; \ldots.$$

Observe that this object satisfies termination (and thus yields well-defined return values), because, with probability 1, some C_i eventually produces agreement, causing the following A_i to force every process to commit.

Suppose now that every process commits in some prefix $A_{-1}; A_0; C_1; A_1; \ldots; C_k; A_k$. Then this prefix is a weak consensus object by Theorem 4, so the output values are consistent with validity and coherence (and thus agreement because the processes commit). It follows that object U is a randomized consensus object.

The initial prefix $A_{-1}; A_0$ implements a fast path for the case where some process p finishes A_{-1} before any process with a different input arrives.[3] If this case holds, then the convergence condition implies that p must commit, because it cannot distinguish this execution from one in which all processes have the same input. But then coherence implies that all processes have the same output from A_{-1} and thus commit in A_0. This avoids the overhead of running a conciliator when conciliators are more expensive than adopt-commit objects.

In a general execution, the cost of object U depends on the cost of the A_i and C_i. Let T be a time measure that is additive (the cost of executing an operation X followed by an operation Y is $T(X) + T(Y)$) and does not charge for internal operations of processes; note that both the individual work $T_{\text{individual}}$ and the total work T_{total} have these properties. The expected value of i at which some C_i successfully produces agreement is at most $1/\delta$, so we have $\mathrm{E}[T(U)] \leq 2T(A) + (1/\delta)(T(C) + T(A)) = O(T(C) + T(A))$ when δ is constant. So the cost of consensus will be asymptotically equal to the worse of the costs of a conciliator and an adopt-commit object.

4.1.2 Bounded construction

The preceding construction requires unbounded space. We can avoid this by leveraging some bounded-space randomized consensus protocol to truncate the sequence of objects (e.g. the bounded-space protocol of [5], which uses polynomial total work and thus also polynomial individual work).

Let A_i and C_i be as above and let $B = (A_{-1}; A_0; C_1; A_1; C_2; A_2; \ldots; C_k; A_k; K)$, where K is a bounded-space randomized consensus protocol. That B is a consensus object follows from Theorem 4 and the fact that K commits if no earlier object does. The expected complexity of B for an additive time measure T is bounded by $O((1/\delta)(T(A) + T(C)) + (1-\delta)^k T(K))$. If δ is constant and $T(K)$ is polynomial in n, then for some $k = O(\log n)$ this reduces to $O(T(A) + T(C))$ as in the previous case.

We state this result as a theorem:

Theorem 5 *Given any bounded-space implementations of conciliators $\{C_i\}$ with constant agreement probability and adopt-commit objects $\{A_i\}$, there exists an implementation of consensus that uses bounded space, with expected individual work $O(\max(T_{\text{individual}}(C_i), T_{\text{individual}}(A_i)))$ and expected total work $O(\max(T_{\text{total}}(C_i), T_{\text{total}}(A_i)))$.*

4.2 Consensus with adopt-commit objects only

Under severe restrictions on the adversary, it is possible to solve consensus without using conciliators at all. Let $A^* = A_1; A_2; \ldots$ consist of an unbounded sequence of adopt-commit objects. If during an execution of A^*, some process p completes A_i before any process with a conflicting value enters A_i, then p commits by the same analysis as for the fast-path prefix above. So if we can guarantee that this eventually happens, A^* implements a consensus protocol.

For binary consensus using a constant-individual-work adopt-commit object, A^* is essentially equivalent to the LEAN-CONSENSUS protocol of [6], so the analysis there shows that A^* will terminate

[3] I am indebted to Azza Abouzeid for suggesting this idea.

in $O(\log n)$ individual work with a **noisy scheduler**. This is a scheduler that chooses in advance the timing of all steps of the algorithm, but has its choices perturbed by random errors that accumulate over time. The proof in [6] shows that eventually this cumulative error will push some process ahead of all the others. We expect that comparable results can be obtained for m-valued consensus, but as our current m-valued adopt-commit object requires $\Theta(\log m)$ work, additional analysis would be needed.

The A^* protocol also works with priority-based scheduling as described by [30]. Here each process is assigned a unique priority that does not change over the duration of its execution of the consensus protocol, and each step is taken by the highest-priority process that currently has a pending operation. It is easy to see that in this model, the highest-priority process to execute the protocol will eventually overtake all other processes and reach some adopt-commit object alone, unless a decision occurs earlier. In either case, we achieve consensus. (The same argument shows that this protocol achieves obstruction-free consensus for any scheduling policy, since obstruction-free consensus requires termination only when some process eventually runs by itself.) The A^* protocol is less efficient than the protocol from [30], which uses only two registers and terminates in at most six operations per process. Part of the reason for this inefficiency is that we are assuming that A is an arbitrary adopt-commit object. Particular implementations of adopt-commit objects might offer better performance.

5 Implementing a conciliator

Here we show how to implement a conciliator. We first show that the classic weak shared coin approach of [11] fits in our framework, and then give a new conciliator for the probabilistic-write model with very strong performance bounds.

5.1 Conciliators from weak shared coins

A **weak shared coin** [11] is a protocol in which each process outputs a bit, and, for some agreement probability $\delta > 0$, it holds that the probability that all processes output 0 and the probability that all processes output 1 are both at least δ, regardless of the adversary's choices. Conciliators are generally not weak shared coins: while a conciliator produces a common output value with some probability, the adversary may have complete control over that value. In the other direction, while weak shared coins do guarantee agreement with some probability, they are generally not conciliators, because they don't guarantee validity. Fortunately, validity is easily enforced.

```
   shared data:
     binary registers r_0 and r_1, initially 0;
     weak shared coin SharedCoin
1  r_v ← 1
2  if r_¬v = 1 then
3      return (adopt, SharedCoin())
4  else
5      return (adopt, v)
6  end
```

Procedure `CoinConciliator(`v`)`

Code for a shared-coin-based binary conciliator is given as Procedure `CoinConciliator`.

Theorem 6 *Given a weak shared coin* `SharedCoin` *with agreement probability δ, Procedure* `CoinConciliator` *satisfies termination, validity, coherence, and probabilistic agreement with agreement probability at least δ.*

Proof: Termination follows from termination of `SharedCoin` and the lack of loops. Validity follows from the fact that if all inputs are v, then no process writes $r_{\neg v}$ and all processes skip the shared coin. Coherence is satisfied vacuously.

For probabilistic agreement, essentially the same argument as used in [11] applies. If every process executes the shared coin, they all return the same value v with probability at least δ for each v. Alternatively, if some process p skips the shared coin and returns v, then it must have written r_v before reading 0 from $r_{\neg v}$. It follows that any process with input $\neg v$ reads r_v after writing $r_{\neg v}$, sees 1, and executes the shared coin. So only v can be returned by processes skipping the shared coin, and with probability at least δ, all processes executing the shared coin also return v. ∎

The cost of this implementation is likely to be dominated by the cost of `SharedCoin`; the other parts add 2 registers and 2 register operations. Note that this implementation only provides a 2-valued conciliator. How to extend a shared coin to more values is not obvious; for one choice, see [20]. Our new algorithm below avoids this restriction to a bounded set of values, at the cost of requiring a weaker adversary.

5.2 Conciliators using probabilistic writes

In the probabilistic-write model, we can build a conciliator for arbitrarily many values, following the approach of the classic algorithm of Chor, Israeli, and Li [23] and its more recent optimization by Cheung [22]. The basic idea is to have a single multi-writer register, which in an ideal execution is written only once by some winning process. This is enforced by having each process attempt to write the register, with small probability, only if it has not yet observed a value in the register. Assuming the probabilities are set correctly, there will be a constant chance that some process writes the register but the next $n-1$ processes do not, causing them to read the winning value and return it.

Previous protocols in this model have used a constant $\Theta(1/n)$ probability for each write. This gives a bound on both total and individual work of $O(n)$. We generalize this approach to allow processes to become impatient over time and increase their probabilities (analogously to the increasing weighted votes in the weak shared-coin protocols of [9, 10, 12]). This new approach will allow us to get optimal $O(n)$ expected total work while reducing the individual work to $O(\log n)$.

Code for the conciliator is given in Procedure `ImpatientFirstMoverConciliator`. Each process repeatedly checks if the register is empty, and if so, writes its input to the register with probability $\frac{2^k}{2n}$, where k is the number of previous failed attempts. If a process observes a value in the register, it returns that value.

Because the process writes with probability 1 once 2^k reaches $2n$, the loop is never executed for more than $\lceil \lg n \rceil + 1$ iterations, at the cost of two operations per iteration (one read and one probabilistic write). Following the last iteration, an additional read is needed to detect $r \neq \perp$.

```
   shared data: register r, initially ⊥
1  k ← 0
2  while r = ⊥ do
3      write v to r with probability 2^k/2n
4      k ← k + 1
5  end
6  return (adopt, r)
```

Procedure `ImpatientFirstMoverConciliator(`v`)`

This gives a bound of $2\lceil \lg 2n \rceil + 3 = 2\lceil \lg n \rceil + 5$ on the individual work.[4] At the same time, since each write succeeds with probability at least $\frac{1}{2n}$, we retain the $O(n)$ bound of previous first-mover protocols on the expected total work. But it may be that the cost of impatient processes is that we lose the constant agreement probability. That this is not the case is shown in the following theorem:

Theorem 7 *Procedure* `ImpatientFirstMoverConciliator` *satisfies termination in expected* $O(n)$ *total work and at most* $2\lg n + O(1)$ *individual work; validity; coherence; and agreement with probability at least* $(1 - e^{-1/4})(1/4) \approx 0.0553$.

Proof: The bounds on individual work and total work have already been established. Validity and coherence are immediate from inspection of the algorithm.

For agreement, observe that the adversary's choices are effectively limited to choosing the order in which the processes will attempt their probabilistic writes, and that it succeeds only if it can get one of the remaining $n-1$ processes to carry out a successful write after some initial process successfully writes to r. Fix some adversary strategy, and let p_i be the probability that the i-th write succeeds if no previous write succeeds. Let t be the minimum value for which $\sum_{i=1}^{t} p_i \geq 1/4$. We will argue that with at least constant probability, some write succeeds in the first t attempts, and thereafter no process writes r on its next attempt. In this case exactly one value is written to r and this value is returned by all processes.

The probability that none of the first t writes succeed is given by $\prod_{i=1}^{t}(1-p_i) \leq \prod_{i=1}^{t} \exp(-p_i) = \exp\left(-\sum_{i=1}^{t} p_i\right) \leq e^{-1/4}$. So with probability at least $1 - e^{-1/4}$, some process writes r at a time t' when $\sum_{i=1}^{t'-1} p_i$ is still small.

The reason we care about this is that we can use the bound on $\sum p_i$ to get a bound on the probability that some other process then overwrites r. Consider any single process p, and suppose that it has failed to carry out its first k_p writes. The probability that its next write succeeds is $\frac{1}{2n}2^{k_p} = \frac{1}{2n}\left(1 + \sum_{j=0}^{k_p-1} 2^j\right) = \frac{1}{2n} + \sum_{j=0}^{k_p-1} \frac{2^j}{2n}$.

If process q carries out the first successful write as one of the first t writes, we can bound the right-hand side of the above equation when summed over all $p \neq q$ by letting each k_p count the number of unsuccessful writes carried out by p among the at most $t-1$ writes preceding q's write.

[4] We assume here that each probabilistic write costs 1 unit whether or not the write succeeds. We do not require that a process detect if it performs a successful write, as it will leave the loop following its next read in any case. If we can detect success, the individual work bound can be reduced by 1 by returning immediately after a successful write.

The probability that q's value is overwritten is then bounded by

$$\begin{aligned}\sum_{p\neq q}\frac{2^{k_p}}{2n} &= \sum_{p\neq q}\left(\frac{1}{2n}+\sum_{j=0}^{k_p-1}\frac{2^j}{2n}\right)\\ &\leq \frac{n-1}{2n}+\sum_{i=1}^{t-1}p_i\\ &< \frac{1}{2}+\frac{1}{4}=\frac{3}{4}.\end{aligned}$$

Note that this bound does not depend on which write succeeds. We thus have

$$\begin{aligned}\Pr[\text{only one write occurs}] &\geq \left(1-e^{-1/4}\right)\left(1-\frac{3}{4}\right)\\ &= \left(1-e^{-1/4}\right)(1/4)\\ &\approx 0.0553.\end{aligned}$$

∎

6 Implementing an adopt-commit object

Here we show how to build a deterministic m-valued adopt-commit object with $\Theta(\log m)$ individual work. This is an exponential improvement on the $\Theta(m)$ individual work of the previously-known adopt-commit implementation based on collect [25].

6.1 Write and read quorums

The operation of our adopt-commit object is similar to the adopt-commit protocol of Gafni [25], and also resembles the follow-the-leader mechanisms in the classic Chor-Israeli-Li [23] and Aspnes-Herlihy [11] consensus protocols. The main difference from this previous work is that we exploit the power of multi-writer registers to reduce the cost.

A process first announces that it has a particular value v by writing to all registers in a **write quorum** W_v that depends on v. It then examines a special *proposal* register proposal. If this register is empty, the process may propose its value by writing to the proposal register; otherwise, it adopts the previously proposed value as its new preferred value in place of v. Whatever value preference it obtains, it returns (commit, preference) only if no other value has been announced; otherwise, it returns (adopt, preference). Conflicting values are detected by reading all registers in a read quorum $R_{\mathsf{preference}}$. This works as long as for every two distinct values v and v', $R_{v'}$ includes some register written in W_v that is not written in $W_{v'}$.

Code for this implementation is given as Procedure `AdoptCommit`.

Theorem 8 *Let $W_v \cap R_{v'} = \emptyset$ if and only if $v = v'$. Then Procedure* `AdoptCommit` *satisfies termination, validity, coherence, and convergence.*

```
    shared data:
      register proposal, initially ⊥;
      binary registers r_i, initially 0
    local data: preference, u
 1  foreach r_i ∈ W_v do
 2      r_i ← 1
 3  end
 4  u ← proposal; if u ≠ ⊥ then
 5      preference ← u
 6  else
 7      preference ← v
 8      proposal ← preference
 9  end
10  if r_i ≠ 0 for some r_i ∈ R_preference then
11      return (adopt, preference)
12  else
13      return (commit, preference)
14  end
```

Procedure `AdoptCommit(`v`)`

Proof: Termination is immediate from the lack of unbounded loops. Validity holds because any return value is either an initial input v or an input read indirectly from proposal. Convergence holds because if every process has input v, then no process announces any value $v' \neq v$, or writes any $v' \neq v$ to proposal, so preference $= v$ for all processes and the test in Line 10 always fails.

For coherence, suppose that some process p returns (commit, v) (it is not hard to see that a process can only return its own input in this case). Then p observed no conflicting v' when it executed Line 10. It follows that no process p' with a conflicting value v' had yet completed Line 3 before p finished executing all of the code up to Line 10. Since p either reads v from proposal in Line 4 or sets proposal to v in Line 8, we have that no process p' with input $v' \neq v$ finishes Line 3 before proposal is set to v. Thus no such p' writes any $v' \neq v$ to proposal, and no process chooses a preference different from v before Line 10. Hence no process outputs a value different from v. ∎

6.2 Choice of quorums

It remains only to specify a choice of quorums satisfying the condition in Theorem 8. Because the cost of Procedure `AdoptCommit` is dominated by writing W_v and checking $R_{\mathsf{preference}}$, our goal is to keep both write quorums and read quorums as small as possible.

Here are some possible choices:

1. For a binary adopt-commit object, use two 1-bit registers r_0 and r_1, and let $W_v = \{r_v\}$ and $R_v = \{r_{\neg v}\}$. Using this implementation, `AdoptCommit` requires at most 4 register operations and uses only 3 registers: r_0, r_1, and proposal.

2. A generalization of the preceding mechanism to m values is to use a pool of k registers $r_1 \ldots r_k$, where $k = \lg m + O(\log \log m)$ satisfies $\binom{k}{\lfloor k/2 \rfloor} \geq m$. For each value v, assign a distinct write

quorum W_v of size $\lfloor k/2 \rfloor$ and a complementary read quorum $R_v = \overline{W}_v$. Then $R_v \cap W_v = \emptyset$ and $R_v \cap W_{v'} \neq \emptyset$ when $v \neq v'$. An m-valued instance of `AdoptCommit` constructed using this technique requires $\lg m + O(\log \log m)$ registers and $\lg m + O(\log \log m)$ individual work.

This choice of quorums is optimal, in the sense of maximizing the number of distinct values m given a fixed bound k on $|W_v| + |R_v|$. This follows from a classic result in extremal set theory, known as Bollabás's Theorem [19]; the version we give here is taken from [27].

Theorem 9 ([27], Theorem 9.8) *Let $A_1, \ldots, A_m$ and $B_1, \ldots, B_m$ be two sequences of sets such that $A_i \cap B_j = \emptyset$ if and only if $i = j$. Then*

$$\sum_{i=1}^{m} \binom{a_i + b_i}{a_i}^{-1} \leq 1, \tag{1}$$

where $a_i = |A_i|$ and $b_i = |B_i|$.

Letting each A_i correspond to some W_i and each B_i to some R_i, each term in the left-hand side of (1) is minimized by setting $|W_i| = \lfloor k/2 \rfloor$; this gives $m = \binom{k}{\lfloor k/2 \rfloor}$ as above.

3. An alternative encoding of values into write quorums gives a simpler implementation with almost as good performance. Here we use a two-dimensional array of registers r_{ij} where $i \in \{1 \ldots \lceil \lg m \rceil\}$ and $j \in \{0, 1\}$. Writing each v as a $\lceil \lg m \rceil$-bit vector, let W_v equal $\{r_{iv_i}\}$ and R_v equal its complement. The rest of the analysis is the same as in the previous method; the space complexity is exactly $2 \lceil \lg m \rceil + 1$ registers and the individual work is at most $2 \lceil \lg m \rceil + 2$ operations.

4. Finally, in a cheap-snapshot or cheap-collect model (where reading an array of n single-writer registers takes $O(1)$ time), we can simulate write quorums of size 1 (and corresponding read quorums of size $m - 1$) by having each process announce its value by writing its own register and having each process test if any process has announced a conflicting value using a single collect operation. The individual work bound in this case is 4 operations, as in the binary case. (While this model is not realistic, a cheap-collect adopt-commit object helps put a limit on what lower bounds can be achieved in a cheap-collect model.) With standard collects implemented as $n - 1$ single-register reads, the same approach costs $\Theta(n)$ individual work, and gives precisely the original adopt-commit protocol of Gafni [25].

Using either construction of $O(\log m)$-sized quorums with Theorem 8 gives the following result:

Theorem 10 *For any m, an m-valued adopt-commit object can be implemented deterministically for any number of processes using $O(\log m)$ atomic registers and $O(\log m)$ individual work.*

7 Discussion

We have shown that separating consensus into explicit objects responsible for generating and detecting agreement allows for a simpler description of the overall protocol and for optimizing the resulting objects independently. For the probabilistic-write model, this gives the first known algorithm with sublinear individual work and the first algorithm with linear total work when only a constant number of input values are permitted.

Considering conciliators and adopt-commit objects separately also offers guidance for seeking possible non-trivial lower bounds on expected work in weak-adversary models with uncorrelated local coins. For m-valued consensus, we currently have two obstacles to improved individual work: we would need to reduce both the $O(\log n)$ upper bound on conciliators and the $O(\log m)$ upper bound on adopt-commit objects. While it is not necessarily the case that any consensus protocol separates cleanly into these components (for example, the protocol of [5] clearly does not), because a consensus object satisfies the specifications of both a conciliator and an adopt-commit object, any lower bound on either object also gives a lower bound on consensus. Concentrating on these more restricted objects may make finding such a lower bound easier, by giving insight into precisely what aspects of consensus make it hard.

Acknowledgments

The decomposition of consensus into conciliator and adopt-commit objects was inspired in part by a desire to give a better unified presentation of known consensus protocols. I would like to thank the students in Yale's Spring 2010 Theory of Distributed Systems class for serving as test subjects for the resulting pedagogical experiment and for offering several helpful comments, and Azza Abouzeid in particular for suggesting that the consensus protocols of Section 4.1 should include a fast path that avoids conciliators when the fastest processes agree. I would also like to thank Keren Censor-Hillel for many useful comments on an early draft of this paper, and Faith Ellen for pointing out the connection between ratifiers, as defined in an earlier version of this paper [8], and the adopt-commit protocols of [25].

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] J. H. Anderson, R. Jain, and D. Ott. Wait-free synchronization in quantum-based multiprogrammed systems. In *Distributed Computing; 12th International Symposium; Proceedings*, volume 1499 of *Lecture Note in Computer Science*, pages 34–45, Andros, Greece, September 1998. Springer-Verlag.

[4] James H. Anderson and Mark Moir. Wait-free synchronization in multiprogrammed systems: Integrating priority-based and quantum-based scheduling. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 123–132, Atlanta, Georgia, USA, 3–6 May 1999.

[5] James Aspnes. Time- and space-efficient randomized consensus. *Journal of Algorithms*, 14(3):414–431, May 1993.

[6] James Aspnes. Fast deterministic consensus in a noisy environment. *Journal of Algorithms*, 45(1):16–39, October 2002.

[7] James Aspnes. Randomized protocols for aynchronous consensus. *Distributed Computing*, 16(2-3):165–176, September 2003.

[8] 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.

[9] James Aspnes, Hagit Attiya, and Keren Censor. Randomized consensus in expected $O(n \log n)$ individual work. In *PODC '08: Proceedings of the Twenty-Seventh ACM Symposium on Principles of Distributed Computing*, pages 325–334, August 2008.

[10] James Aspnes and Keren Censor. Approximate shared-memory counting despite a strong adversary. *ACM Transactions on Algorithms*, 6(2):1–23, 2010.

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

[12] James Aspnes and Orli Waarts. Randomized consensus in expected $O(N \log^2 N)$ operations per processor. *SIAM Journal on Computing*, 25(5):1024–1044, October 1996.

[13] Hagit Attiya and Keren Censor. Tight bounds for asynchronous randomized consensus. *Journal of the ACM*, 55(5):20, October 2008.

[14] Hagit Attiya and Keren Censor-Hillel. Lower bounds for randomized consensus under a weak adversary. *SIAM J. Comput.*, 39(8):3885–3904, 2010.

[15] 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.

[16] Yonatan Aumann and Michael A. Bender. Efficient low-contention asynchronous consensus with the value-oblivious adversary scheduler. *Distributed Computing*, 17(3):191–207, 2005.

[17] László Babai. Monte-carlo algorithms in graph isomorphism testing. Technical Report D.M.S. 79-10, Université de Montréal, 1979.

[18] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 27–30, Montreal, Quebec, Canada, August 1983.

[19] B. Bollobás. On generalized graphs. *Acta Mathematica Hungarica*, 16(3):447–452, September 1965.

[20] Keren Censor-Hillel. Multi-sided shared coins and randomized set-agreement. In *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallel Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010*, pages 60–68, June 2010.

[21] 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.

[22] 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.

[23] Benny Chor, Amos Israeli, and Ming Li. Wait-free consensus using asynchronous hardware. *SIAM J. Comput.*, 23(4):701–712, 1994.

[24] 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.

[25] 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.

[26] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[27] Stasys Jukna. *Extremal combinatorics: with applications in computer science*. Springer-Verlag, 2001.

[28] Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, pages 163–183, 1987.

[29] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[30] Srikanth Ramamurthy, Mark Moir, and James H. Anderson. Real-time object sharing with minimal system support. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 233–242, New York, NY, USA, 1996. ACM.