Multi-Sided Shared Coins and Randomized Set-Agreement

Keren Censor Hillel*
Department of Computer Science
Technion
Haifa 32000, Israel
ckeren@cs.technion.ac.il

ABSTRACT

This paper presents wait-free randomized algorithms for solving *set-agreement* in asynchronous shared-memory systems under a *strong adversary*. First, the definition of a shared-coin algorithm is generalized to a *multi-sided shared-coin* algorithm, and it is shown how to use any multi-sided shared coin in order to obtain a randomized set-agreement algorithm for agreeing on k values out of $k+1$. Then, an implementation is given for a $(k+1)$-sided shared coin for n processes with a constant agreement parameter, $O(n^2/k)$ total step complexity, and $O(n/k)$ individual step complexity. This implementation yields a randomized set-agreement algorithm for agreeing on k values out of $k+1$ with a total step complexity of $O(n^2/k + nk)$ and an individual step complexity of $O(n/k + k)$. Next, other set-agreement algorithms for agreeing on ℓ values out of $k+1$, where ℓ is smaller than k, are presented. This includes the case of *multi-valued consensus* in which $\ell = 1, k > 1$. To the best of our knowledge, these are the first wait-free algorithms for set-agreement in the asynchronous shared-memory model under a strong adversary that are not for the specific case of binary consensus, where $\ell = k = 1$. Finally, an application of asynchronous wait-free multi-valued consensus is presented, in implementing *at-most-once* semantics with optimal effectiveness.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent programming*; F.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity—*Nonnumerical Algorithms and Problems*; G.3 [**Mathematics of Computing**]: Probability and Statistics—*Probabilistic algorithms*

General Terms

Algorithms, Theory

*Supported in part by the Adams Fellowship Program of the Israel Academy of Sciences and Humanities and by the *Israel Science Foundation* (grant number 953/06).

SPAA'10, June 13–15, 2010, Thira, Santorini, Greece.

Keywords

distributed computing, shared memory, randomized algorithms, set-agreement, multi-valued shared coins

1. INTRODUCTION

The problem of *set-agreement* was introduced by Chaudhuri [9] as a generalization of the consensus problem, in order to overcome the well-known impossibility of solving consensus deterministically in an asynchronous system which allows even one crash-failure [13]. In the set-agreement problem, n processes start with input values in $\{0, \ldots, k\}$ and should produce output values such that there are at most ℓ different outputs, for some $\ell < n$. The termination condition requires every non-faulty process to eventually decide, and to avoid trivial solutions, the validity condition requires each output value to be the input value of some process.

Chaudhuri showed that if the bound f on the number of faulty processes is smaller than ℓ, then set-agreement can be solved by a deterministic algorithm. Later, it was shown by Borowsky and Gafni [7], Herlihy and Shavit [14], and Saks and Zaharoglou [20], using topological arguments, that set-agreement cannot be solved deterministically in an asynchronous system if $f \geq \ell$, and in particular it does not have a wait-free solution (i.e., $f = n - 1$, implying that it may be that all but one process fail).

Another approach to overcome the FLP impossibility result for consensus, is to relax the termination condition to hold only with probability 1, and thus allow the use of randomization. Many randomized consensus algorithms were designed for different models of communication and timing assumptions. As in the case of consensus, randomization also allows to overcome the impossibility result for set-agreement.

In this paper, we present randomized wait-free algorithms for solving set-agreement in an asynchronous shared-memory system. First, in Section 3, we generalize the definition of a *shared-coin* algorithm, and define *multi-sided shared-coin* algorithms. In such an algorithm, each process outputs one of $k+1$ values (instead of one of two values as in a regular shared-coin), such that each subset of k values has probability at least δ for containing the outputs of all the processes. In other words, each value has probability at least δ of *not* being the output of any process. We then extend the Aspnes-Herlihy framework for using a shared coin for obtaining a randomized consensus algorithm [4], and show how to use any multi-sided shared coin in order to obtain a randomized set-agreement algorithm, for agreeing on k values out of $k+1$.

Next, in Section 4, we present an implementation of a $(k+1)$-sided shared-coin algorithm which has a constant agreement parameter, $O(n^2/k)$ total step complexity, and $O(n/k)$ individual step complexity. We then derive a set-agreement algorithm from the $(k+1)$-sided shared coin using the above framework.

Algorithm	Parameters	Method	Individual Step Complexity	Total Step Complexity
Section 3	$k, k+1$	multi-sided shared coin	$O(n/k + k)$	$O(n^2/k + nk)$
Section 5.1	$\ell, k+1$	space reduction	$O(n(\log k - \log \ell))$	$O(n^2(\log k - \log \ell))$
Section 5.2	$\ell, k+1$	iterative	$O((k-\ell+1)k + n(\log k - \log \ell))$	$O((k-\ell+1)nk + n^2(\log k - \log \ell))$
Section 5.3	$1, k+1$	bit-by-bit	$O(n \log k)$	$O(n^2 \log k)$

Figure 1: The agreement algorithms presented in this paper.

In Section 5, we present set-agreement algorithms that are designed for agreeing on ℓ values out of $k+1$, for $\ell < k$. In particular, they can be used for the case $\ell = 1$, where the processes agree on the same value, i.e., for *multi-valued consensus*. By definition, solving multi-valued consensus is at least as hard as solving *binary consensus* (where the inputs are in the set $\{0, 1\}$, i.e., $k = 1$), and potentially harder. One algorithm uses multi-sided shared coins, while the other two embed binary consensus algorithms in various ways.

To the best of our knowledge, these are the first wait-free algorithms for set-agreement in the shared-memory model under a strong adversary, other than binary consensus. Figure 1 shows the properties of the different algorithms we present. For $\ell < k$ one of our algorithms is better than the others; however, intrigued by the question of whether multi-valued consensus is inherently harder than binary consensus, we find the different methods interesting in hope that one of them could lead to a lower bound.

Finally, we show in Section 6 an application of asynchronous wait-free multi-valued consensus in implementing *at-most-once* semantics with optimal effectiveness. At-most-once semantics [15] requires m jobs to be executed with the guarantee that no job is performed more than once. A trivial solution simply does not execute any job. However, the *effectiveness* of an algorithm for solving the at-most-once problem is the number of *completed* jobs, and it is desired to find algorithms with the maximal effectiveness possible. Section 6 shows how to use a randomized multi-valued consensus algorithm in order to solve the at-most-once problem while obtaining the optimal effectiveness, thus answering an open question raised in [15].

Related Work:. Previous randomized agreement algorithms for asynchronous shared-memory systems under a strong adversary are for the specific case of binary consensus (e.g., [1–5, 8, 19]). The optimal individual and total step complexities are $O(n)$ and $O(n^2)$, respectively [3, 5].

Unlike the shared-memory model, several set-agreement algorithms for asynchronous *message-passing* systems have been proposed. Mostefaoui et al. [17] use binary consensus to construct a multi-valued consensus algorithm for message-passing systems. This work assumes reliable broadcast. In the same model, Zhang and Chen [23] present improved algorithms, which reduce the number of binary consensus instances that are required. Under the above assumption, Ezhilchelvan et al. [12] also present a randomized multi-valued consensus algorithm, while Mostefaoui and Raynal [16], present a randomized set-agreement algorithm for agreeing on ℓ values out of n. The above algorithms require a bound on the number of failures $f < n/2$, a restriction that can be avoided in the shared-memory model. Moreover, there is an exponentially small agreement probability for the coins that are used, which causes the expected number of phases until agreement is reached to be large.

There is additional literature on set-agreement in *synchronous* systems. Chaudhuri et al. [10] show that $f/k + 1$ is the number of rounds needed for solving set-agreement in shared memory. Turpin and Coan [22] propose a set-agreement algorithm for message passing with Byzantine failures that embeds consensus to agree on the credibility of sent messages. Their algorithm satisfies a weaker validity condition, which requires that if all input values are the same, then every output value is that input value. Raynal and Travers [18] propose new algorithms in addition to a good survey on synchronous set-agreement algorithms.

2. MODEL AND DEFINITIONS

We consider a standard model of an asynchronous system in which n processes $\{p_0, \ldots, p_{n-1}\}$ communicate by reading and writing to shared multi-writer multi-reader registers. A *step* of a process consists of one access to the shared memory, followed by local computation and/or local coin-flips. Processes may fail by crashing, in which case they do not take any further steps. We require our algorithms to be *wait-free*, i.e., to be correct even if $n-1$ processes may fail during an execution. The system is *asynchronous*, meaning that the steps of processes are scheduled according to an adversary. This implies that there are no timing assumptions, and specifically no bounds on the time between two steps of a process, or between steps of different processes.

For completeness, we formally define the problem of set-agreement as follows. In an algorithm for solving $(\ell, k+1, n)$-*agreement* each process p_i starts with an input value in $\{0, \ldots, k\}$ and should produce an output value such that the following conditions hold:

- Set-Agreement: there are at most ℓ different outputs.
- Validity: every output is the input of some process.
- Termination: every non-faulty process eventually decides.

We sometimes use the term *set-agreement* without parameters for abbreviation. The particular case in which $\ell = 1, k > 1$ is the problem of *multi-valued consensus*, while in case $\ell = k = 1$ we have *binary consensus*.

A *randomized* algorithm for set-agreement is required to satisfy a relaxed termination condition: *with probability 1*, every non-faulty process eventually decides.

We measure the *individual step complexity* of a randomized set-agreement algorithm as the expected number of steps taken by any single process. Similarly, the *total step complexity* is the expected number of steps taken by all the processes.

In Section 3 we present a framework for randomized algorithms which solve $(k, k+1, n)$-agreement using a *multi-sided shared-coin* algorithm. We now formally define such a procedure, which is a generalization of a shared coin (which in our terms is a 2-sided shared coin). A $(k+1)$-*sided shared-coin* algorithm with *agreement parameter* δ is an algorithm in which every non-faulty process

p produces an output value in $\{0, \ldots, k\}$, such that for every subset of size k there is probability at least δ that all the outputs are within that subset. Alternatively, for every value v in $\{0, \ldots, k\}$ there is probability at least δ that v is *not* the output of any process. We emphasize that unlike the requirement of set-agreement, the probability of disagreement in a shared coin may be greater than 0. Notice that there are no inputs to this procedure.

Since our algorithms are randomized, different assumptions on the power of the adversary may yield different results. Throughout this work, we assume a *strong adversary*. Such an adversary can base its next scheduling decision on the local state of all the processes, including the results of local coin-flips. Notice, however, that the adversary does not know the results of local coins that were not yet flipped[1].

3. A $(k, k+1, n)$-AGREEMENT ALGORITHM USING A $(k+1)$-SIDED SHARED COIN

In this section we present a framework for randomized $(k, k+1, n)$-agreement algorithms. It is a generalization of the framework of Aspnes and Herlihy [4] for deriving a randomized binary consensus algorithm from a weak shared coin, and specifically follows the presentation given by Saks, Shavit, and Woll [19]. However, its complexity is improved by using multi-writer registers, based on the construction of Cheung [11].

We assume a $(k+1)$-sided shared-coin algorithm called sharedCoin$_{k+1}$, with an agreement parameter δ_{k+1}. The set-agreement algorithm is given in Algorithm 1. Throughout the paper, we assume that shared arrays are initialized to a special symbol $\perp$. Informally, the set-agreement algorithm proceeds by (asynchronous) phases, in which each process p writes its own preference to a shared array $Propose$, checks if the preferences agree on k values, and notes this in another shared array $Check$. If p indeed sees agreement, it also notes its preference in $Check$.

Process p then checks the agreement array $Check$. If p does not observe a note of disagreement, it decides on the value of its preference. Otherwise, if there is a note of disagreement, but also a note of agreement, p adopts the value associated with the agreement notification as preference for the next phase. Finally, if there is only a notification of disagreement, the process participates in a $(k+1)$-sided shared-coin algorithm and prefers the output of the shared coin.

LEMMA 1. *Consider a phase $r \geq 1$ and a non-faulty process p that finishes phase r. If all the processes that start phase r before p finishes it have at most k preferences in $\{v_1, \ldots, v_k\}$, then p decides $v \in \{v_1, \ldots, v_k\}$ in this phase r.*

PROOF. We claim that p reads $Check[r]$[disagree] $==$ false in line 9 of phase r, and therefore decides in phase r. This will also imply that its decision value v is in $\{v_1, \ldots, v_k\}$, otherwise p is among the processes that start phase r before p finishes, but does not have a preference in $\{v_1, \ldots, v_k\}$, which contradicts our assumption. Assume towards a contradiction, that p reads $Check[r]$[disagree] $==$ true in line 9 of phase r. This implies that there is a process q that writes $Check[r]$[disagree] $=$ true in line 7 of phase r, and this happens before p finishes. Therefore, q reads more than k values in $Propose[r]$ in line 3 of phase r, which means that there are $k+1$ processes that write $k+1$ different values to $Propose[r]$ in line 2 of phase r, and all this happens before p finishes. But this contradicts our assumption that all the processes that start phase r before a non-faulty process p finishes it have at most k preferences. $\square$

[1]Thus, if we model local coin-flips as having a random tape for each process, then the adversary knows the content of the tape only in locations that were accessed by the process.

Algorithm 1 A $(k, k+1, n)$-agreement algorithm, code for p_i

```
Local variables: r = 1, decide = false, myValue = input,
   myPropose = [ ], myCheck = [ ]
Shared arrays: Propose[ ][0..k], Check[ ][agree, disagree]
1:  while decide == false
2:    Propose[r][myValue] = true
3:    myPropose = collect(Propose[r])
4:    if the number of values in myPropose is at most k
5:      Check[r][agree] = ⟨true, myValue⟩
6:    else
7:      Check[r][disagree] = true
8:    myCheck = collect(Check[r])
9:    if myCheck[disagree] == false
10:     decide = true
11:   else if myCheck[agree] == ⟨true, v⟩
12:     myValue = v
13:   else if myCheck[agree] == false
14:     myValue = sharedCoin_{k+1}[r]
15:   r = r + 1
16: end while
17: return myValue
```

Lemma 1 implies validity, by applying it for phase $r = 1$. The next two lemmas are used to prove the agreement condition. Below, we use the notation $\langle$true, ?$\rangle$ for an entry in the array $Check$ which has *true* as its first element, and any value as its second element.

LEMMA 2. *For every phase $r \geq 1$, all the processes that read $Check[r][agree] == \langle true, ? \rangle$ and finish phase r have at most k different preferences at the end of phase r.*

PROOF. We first claim that all the processes that write to $Check[r]$[agree] wrote at most k different preferences to $Propose[r]$. Assume, towards a contradiction, that among the processes that write to $Check[r]$[agree] there are $k+1$ processes $\{p_{i_1}, \ldots, p_{i_{k+1}}\}$ that wrote $k+1$ different preferences to $Propose[r]$. Let p_{i_j} be the last process to write to $Propose[r]$. When p_{i_j} collects $Propose[r]$ in line 3, it reads $k+1$ values, and therefore does not write to $Check[r]$[agree], which is a contradiction.

The above claim implies that at most k different preferences may be written to $Check[r]$[agree]. Since a process that reads $Check[r]$[agree] $==$ $\langle$true, $v\rangle$ adopts v as its preference, at most k values can be a preference of such processes at the end of phase r. $\square$

LEMMA 3. *For every phase $r \geq 1$, if processes decide on values in $\{v_1, \ldots, v_k\}$ in phase r, then every non-faulty process decides on a value in $\{v_1, \ldots, v_k\}$ in phase r', where r' is either r or $r+1$.*

PROOF. We first claim that if a process decides v in phase r, then every non-faulty process that finishes phase r reads $Check[r]$[agree] $==$ $\langle$true, ?$\rangle$. To prove the claim, let p be a process that decides v in phase r. Let q be a non-faulty process that finishes phase r, and assume towards a contradiction that q reads $Check[r]$[agree] $==$ false. This implies that q collects $Check[r]$ in line 8 before p writes to $Check[r]$ in line 5, and therefore p collects $Check[r]$ after q writes to $Check[r]$[disagree], which implies that p does not decide in phase r, a contradiction.

Now, let p be a process that decides in phase r, and let q be a non-faulty process. By the above claim, q reads $Check[r][\text{agree}] == \langle \text{true}, ? \rangle$ in line 8. By Lemma 2, there are at most k different values that can become a preference of a process at the end of phase r. Therefore, if q decides at the end of phase r then it decides a value in $\{v_1, \ldots, v_k\}$. Otherwise, all the non-faulty processes write at most k preferences to $Propose[r+1]$, and by Lemma 1, they decide on one of these values at the end of phase $r+1$. □

Lemma 3 implies agreement. Notice that both validity and agreement are *always* satisfied, and not only with probability 1. For termination, we prove the following lemma. Below, we denote the agreement parameter of the $(k+1)$-sided shared coin by $\delta = \delta_{k+1}$.

LEMMA 4. *The expected number of phases until all non-faulty processes decide is at most* $1 + 1/\delta$.

PROOF. For every subset $\{v_1, \ldots, v_k\} \subseteq \{0, \ldots, k\}$ there is a probability of at least δ for all processes that run $\mathsf{sharedCoin}_{k+1}$ to output values in $\{v_1, \ldots, v_k\}$. Therefore, for any value v in $\{0, \ldots, k\}$, there is a probability of at least δ that v is not the output of any process running $\mathsf{sharedCoin}_{k+1}$. This is because $\{0, \ldots, k\} \setminus \{v\}$ has probability of at least δ for containing the outputs of all the processes.

Consider a phase $r \geq 2$. By Lemma 2, all the processes that finish phase $r-1$ and in line 8 read $Check[r-1][\text{agree}] == \langle \text{true}, ? \rangle$ propose at most k values to $Propose[r]$. The other processes propose to $Propose[r]$ a value obtained from their shared coin. Therefore, there is a probability of at least δ that all processes write at most k different values to $Propose[r]$, and by Lemma 1, decide by the end of phase r.

Therefore, after phase $r = 1$, the expected number of phases until all non-faulty processes decide, is the expectation of a geometrically distributed random variable with success probability at least δ, which is at most $1/\delta$.

For the first phase $r = 1$, the values written to $Propose[1]$ are the inputs and are therefore controlled by the adversary. This implies that the expected number of phases until all non-faulty processes decide is at most $1 + 1/\delta$. □

Consider a $(k+1)$-sided shared coin algorithm with an agreement parameter $\delta = \delta_{k+1}$, a total step complexity of $T = T_{k+1}$, and an individual step complexity of $I = I_{k+1}$. In each phase, a process takes $O(k)$ steps in addition to the I steps it takes in the $\mathsf{sharedCoin}_{k+1}$ algorithm. Combining this with Lemma 4, which bounds the expected number of phases until all non-faulty processes decide, gives:

THEOREM 5. *Algorithm 1 solves $(k, k+1, n)$-agreement with $O(\frac{I+k}{\delta})$ individual step complexity and $O(\frac{T+nk}{\delta})$ total step complexity.*

4. A $(k+1)$-SIDED SHARED COIN

We present, in Algorithm 2, a $(k+1)$-sided shared-coin algorithm which is constructed by using k instances of a 2-sided shared coin. We statically partition the processes into k sets of at most $\frac{n}{k}$ processes each. That is, for every j, $0 \leq j \leq k-1$, we have a set $P_j = \left\{ p_{\frac{jn}{k}}, \ldots, p_{\frac{(j+1)n}{k}-1} \right\}$ (for $j = k-1$ the set may be smaller). The processes of each set P_j run a 2-sided shared-coin algorithm $\mathsf{sharedCoin}[j]$ and output the result plus the value j. The idea is that in order to have a value j that is not the output of any process, it is enough that all processes running $\mathsf{sharedCoin}[j-1]$ agree on the value 0 and therefore output $j-1$, and that all the processes running $\mathsf{sharedCoin}[j]$ agree on the value 1 and therefore output $j+1$.

Algorithm 2 A $(k+1)$-sided shared coin algorithm, code for process p_i

Local variables: $j = \lfloor \frac{ik}{n} \rfloor$
1: return $\mathsf{sharedCoin}[j] + j$

Let $\delta = \delta_2$ be the agreement parameter of the 2-sided shared coin. We bound the agreement parameter of the $k+1$-sided shared coin in the next lemma.

LEMMA 6. *Algorithm 2 is a $(k+1)$-sided shared coin with an agreement parameter δ^2.*

PROOF. There is a probability of at least δ for all processes who run $\mathsf{sharedCoin}[j]$ to return the value j, and a probability of at least δ for all processes who run $\mathsf{sharedCoin}[j]$ to return the value $j+1$. Therefore, for any value in $\{0, \ldots, k\}$, there is a probability of at least δ^2 that this value is not the output of any process running $\mathsf{sharedCoin}[j]$, for any $0 \leq j \leq k-1$ (because $j = 0$ may be the output only of $\mathsf{sharedCoin}[0]$, $j = k$ only of $\mathsf{sharedCoin}[k-1]$, and $j \in \{1, \ldots, k-1\}$ only of $\mathsf{sharedCoin}[j-1]$ and $\mathsf{sharedCoin}[j]$). Therefore, Algorithm 2 is a $(k+1)$-sided shared coin with an agreement parameter δ^2. □

The next lemma gives the complexity of the $k+1$-sided shared coin, and follows immediately from the fact that each process runs a 2-sided shared coin algorithm for $\frac{n}{k}$ processes. Since the complexities depend on the number of processes t that may run an algorithm, we now carefully consider this in the notation. Let $I(t) = I_2(t)$ and $T(t) = T_2(t)$ be the individual and total step complexities, respectively, of the 2-sided shared coin with t processes.

LEMMA 7. *Algorithm 2 has individual and total step complexities of $O(I(\frac{n}{k}))$ and $O(k \cdot T(\frac{n}{k}))$, respectively.*

Plugging Lemmas 6 and 7 into Theorem 5 gives:

THEOREM 8. *Algorithm 1 solves $(k, k+1, n)$-agreement with individual step complexity of $O((I(\frac{n}{k})+k)/\delta^2)$ and total step complexity of $O((k \cdot T(\frac{n}{k}) + nk)/\delta^2)$.*

By using an optimal 2-sided shared coin [3] with a constant agreement parameter, an individual step complexity of $O(t)$, and a total step complexity of $O(t^2)$, we get that Algorithm 2 is a $(k+1)$-sided shared coin with a constant agreement parameter, and individual and total step complexities of $O(\frac{n}{k})$ and $O(\frac{n^2}{k})$, respectively. Therefore, Algorithm 1 solves $(k, k+1, n)$-agreement with individual step complexity of $O(\frac{n}{k} + k)$ and total step complexity of $O(\frac{n^2}{k} + nk)$. Note that for $n \geq k^2$, Algorithm 1 has $O(\frac{n}{k})$ individual step complexity, and $O(\frac{n^2}{k})$ total step complexity, which are the same as the complexities of binary consensus divided by k.

5. $(\ell, k+1, n)$-AGREEMENT ALGORITHMS

In this section we construct several algorithms for the solving $(\ell, k+1, n)$-agreement, where $\ell < k$.

5.1 An $(\ell, k+1, n)$-Agreement Algorithm by Space Reduction

For agreeing on one value out of $\{0, \ldots, k\}$ we can get a total step complexity of $O(n^2 \log k)$ by reducing the possible values by half until we have one value. We later show how this construction can be used for agreeing on $\ell > 1$ values.

In Algorithm 3 we assume an array $Agree$ of binary consensus instances, which a process can execute with a proposed value. Algorithm 3 can be modelled as a binary tree, where the processes begin at the leaves, which represent all of the values, and in every iteration j the processes agree on the value of the next node, going up to the root. This means that at most half of the suggested values are decided in each iteration. In addition, all decided values are valid because this is true for each node.

Algorithm 3 A $(1, k+1, n)$-agreement algorithm by space reduction, code for p_i

```
Local variables myValue = input, myPair, mySide
Shared arrays: Agree[1..⌈log (k+1)⌉][1..k/2^j],
   Values[1..⌈log (k+1)⌉][1..k/2^j][0..1]
1:  for j = 1 ... ⌈log (k+1)⌉
2:     myPair = ⌊myValue / 2^j⌋
3:     if myValue − myPair · 2^j < 2^(j−1)
4:        mySide = 0
5:     else mySide = 1
6:     Values[j][myPair][mySide] = myValue
7:     side = Agree[j][myPair](mySide)
8:     myValue = Values[j][myPair][side]
9:  end for
10: return myValue
```

LEMMA 9. *(Validity) For every j, the variable $myValue$ of a process p at the end of iteration j is an input of some process.*

PROOF. The proof is by induction on j. The base case $j = 1$ is clear since $myValue$ is initialized with the input of the process. For the induction step, assume the lemma holds up to $j-1$, and notice that $myValue$ is updated only in line 8, to the value written in the $Values$ array in the location $side$ which is returned from the binary consensus protocol. Since the consensus protocol satisfies validity, $side$ has to be the input of some process to the consensus protocol, and this only happens if that process first writes to that location in the $Values$ array in line 6. By the induction hypothesis, that value is the input of some process. □

LEMMA 10. *(Agreement) Every two process executing Algorithm 3 output the same value.*

PROOF. We claim that there can be at most one value written to $Values[j][pair][side]$, and prove this by induction, where the base case is trivial since at the beginning a process writes to $Values[1][pair][side]$ only if its input value is $2 \cdot pair + side$. Assume this holds up to iteration $j-1$. By the agreement property of the consensus protocol, all processes that execute $Agree[j-1][pair]$ output the same value. Therefore, in iteration j, only one value out of $\{2^j \cdot pair, \ldots, 2^j(pair+1) - 1\}$ can be written to $Values[j][pair][side]$. The lemma follows by applying the claim to the root, which satisfies agreement. □

Termination follows from the termination property of the binary consensus instances. For each j, a process executes one consensus protocol, plus $O(1)$ additional accesses to shared variables. By using an optimal binary consensus protocol where a process completes within $O(n)$ steps, this implies:

THEOREM 11. *Algorithm 3 solves $(1, k+1, n)$-agreement with an individual step complexity of $O(n \log k)$ and a total step complexity of $O(n^2 \log k)$.*

Note that we can backstop this construction at any level j at the tree to get an agreement on $\ell = 2^{\log k - j}$ values. This means that instead of having j iterate from 1 to $\lceil \log(k+1) \rceil$, the algorithm changes so that j iterates from 1 to $\lceil \log(k+1) \rceil - \lceil \log \ell \rceil$. The individual step complexity is $O(n(\log k - \log \ell))$, and the total step complexity is $O(n^2(\log k - \log \ell))$.

5.2 An Iterative $(\ell, k+1, n)$-Agreement Algorithm

In Algorithm 5, we construct an $(\ell, k+1, n)$-agreement algorithm by iterating Algorithm 1 and reducing the number of possible values by one until all processes output no more than ℓ values. The idea is that the processes execute consecutive iterations of $(s, s+1, n)$-agreement algorithms for values of s decreasing from k to ℓ. In each iteration the number of possible values is reduced until it reaches the desired bound ℓ.

This procedure is less trivial than it may appear because, for example, after the first iteration outputs no more than k values out of $k+1$, in order to decide on $k-1$ out of the k values that are possible, the processes need to know *which* are the k possible values. However, careful inspection shows that they need to know these k values only if they disagree upon choosing the $k-1$ values out of them. In this case, a process that sees k values indeed knows which are these values among the initial $k+1$.

The pseudocode appears in Appendix A, as well as the proof of its correctness and complexity, as stated in the following theorem. When using the $(s+1)$-sided shared coins of Section 4 we have:

THEOREM 12. *Algorithm 5 solves $(\ell, k+1, n)$-agreement with $O((k-\ell+1)k + n(\log k - \log \ell))$ individual step complexity and $O((k-\ell+1)nk + n^2(\log k - \log \ell))$ total step complexity.*

5.3 A Bit-By-Bit $(1, k+1, n)$-Agreement Algorithm

For agreeing on one value out of $\{0, \ldots, k\}$ we construct Algorithm 6, which agrees on each bit at a time while making sure that the final value is valid. A similar construction appears in [21, Chapter 9], but does not address the validity condition. In this algorithm, obtaining validity is a crucial point in the construction, since simply agreeing on enough bits does not guarantee an output that is the input of some process.

The idea of our algorithm is that in every iteration j, all the $myValue$ local variables share the same first $j-1$ bits, and they are all valid values (each is the input of at least one process).

The pseudocode appears in Appendix B, as well as the proof of its correctness and complexity, as stated in the following theorem. We denote by $\delta = \delta_2$ the agreement parameter of the 2-sided shared coin, and $T = T_2$ and $I = I_2$ are its total and individual step complexities, respectively.

THEOREM 13. *Algorithm 6 solves $(1, k+1, n)$-agreement with $O(\lceil \log(k+1) \rceil \cdot \frac{I}{\delta})$ individual step complexity and $O(\lceil \log(k+1) \rceil \frac{T}{\delta})$ total step complexity.*

Using an optimal shred coin with a constant agreement parameter, an individual step complexity of $O(n)$, and a total step complexity of $O(n^2)$, we get a $(1, k+1, n)$-agreement algorithm with an individual step complexity of $O(n \log k)$ and a total step complexity of $O(n^2 \log k)$. Notice that the step complexity could be improved if agreement on the bits could be run in parallel. However, this is not trivial because of the need to maintain validity.

6. APPLICATION: THE AT-MOST-ONCE PROBLEM

In addition to the importance of set-agreement as a basic problem in distributed computing, it can also be used to solve other practical problems. In this section, we show how to use a randomized multi-valued consensus algorithm in order to solve the *at-most-once* problem, while improving previous known guarantees for a measure of performance.

The at-most-once semantics ensures that operations in a distributed system occur no more than once. This requirement was studied in contexts of at-most-once message delivery, and of at-most-once remote procedure calls (RPC), and has been recently addressed in the context of asynchronous shared memory by Kentros et al. [15]. In this problem, n processes are required to perform m jobs. Any job may be performed by any process, but it is required that no job is executed more than once. While a trivial solution would be to simply not execute any job, we are interested in algorithms that can nevertheless guarantee some amount of completed jobs.

DEFINITION 1 ([15]). *The* effectiveness *of an algorithm for the at-most-once problem with m jobs, and n processes prone to f crash-failures, is the minimal number of completed jobs over all possible executions.*

Kentros et al. [15] present a deterministic solution for the at-most-once problem with effectiveness $(k-1)^h$, where $m = k^h$ and $n = 2^h$, for wait-free solutions[2]. They also prove a lower bound of $m - f$ on the effectiveness of any such algorithm. While the proof is non-trivial, the intuition is rather simple: a process p may fail just before it is about to execute a certain job, but no other process q can take over that job, since it cannot distinguish this situation and the case where p is simply slow. If q executes the job and p was just slow and eventually also executes the job, the at-most-once semantics is violated. Following this intuition, and although we do not give a formal proof, it is clear that the lower bound of $m - f$ also holds for randomized solutions, which allow termination with probability 1, but require the at-most-once condition to always hold.

Using our randomized wait-free $(1, n, n)$-agreement algorithms (multi-valued consensus out of n values), we show a randomized algorithm for the at-most-once problem, that has optimal effectiveness of $m - f$. The idea is that for every job the processes execute $(1, n, n)$-agreement, and the process whose value was decided is the process that executes the job. A process participates in the multi-valued consensus algorithm for a job only after it finished participating in the multi-valued consensus algorithms for all the previous jobs. In the algorithm we assume m objects of $(1, n, n)$-agreement called $Agreement[1..m]$.

Algorithm 4 An at-most-once algorithm, code for p_i

```
Local variables j
Shared arrays: Agreement[1..m]
1: for j = 1..m
2:     if Agreement[j](i) == i execute job j
```

Algorithm 4 solves the at-most-once problem because of the agreement property of the multi-valued consensus algorithm. Further, in Algorithm 4 a failed process p_i can only block one job j, if i was the value decided upon in $Agreement[j]$, and p_i fails before executing job j. This is because the value i can only be proposed to $Agreement[j]$ by p_i, and therefore by validity can only be decided if p_i invokes $Agreement[j](i)$. This can only happen for one value of j at any given time, by ensuring that a process participates only in one multi-valued consensus algorithm at a time. Therefore we have:

THEOREM 14. *Algorithm 4 solves the at-most-once problem and has effectiveness $m - f$.*

Notice that our algorithm can be easily adapted to the case where the number of jobs is unknown in advance, by simply having an array $jobs$ that indicates whether another job exists, and having the processes check this array before proceeding. Moreover, our solution also works in an online setting of this problem where the jobs arrive during the execution, at times that are controlled by the adversary. This is done by having the processes repeatedly read the first empty location in the $jobs$ array (or linked-list) until they see an indication for a new job.

Another algorithm for the at-most-once problem can use the tree construction of Kentros et al. [15], and add a binary consensus object for every node in the tree, in order to improve the effectiveness of their algorithm. This addition prevents having a location in the array of a node in the tree, which is not addressed by processes of either "side" because of a possibility for a conflict (we refer the reader to the above paper for more details on their algorithm). However, our implementation using multi-valued consensus is more powerful in the sense of being adaptive to the number of jobs that need to be executed.

7. DISCUSSION

This paper presents wait-free randomized algorithms for the set-agreement problem in asynchronous shared-memory systems. There are many open questions that arise and are interesting subjects for further research, as we elaborate next.

We extended the definition of shared-coin algorithms to multi-sided shared coins. It is an open question whether our $(k+1)$-sided shared-coin algorithm can be improved while keeping the agreement parameter constant. In addition, the definition can be modified so that the agreement parameter holds for subsets of less than k values. It is interesting to find good implementations for multi-sided shared coins that satisfy this modified definition.

For randomized set-agreement algorithms, it is open whether better algorithms exist in this model. In addition, it would be intriguing to prove lower bounds on the complexities of such algorithms, as no such bounds are currently known.

We note that for $k \leq \sqrt{n}$ the total and individual step complexities of our $(k, k+1, n)$-agreement algorithm are the same as for the optimal algorithm for randomized binary consensus, only divided by k (see [5] and [3] for the total and individual step complexities of randomized consensus, respectively). First, it is an open question whether the same complexities can be obtained for larger values of k. In addition, a similar relation between consensus and set-agreement occurs also for complexities in deterministic synchronous algorithms and lower bounds under f failures, since the optimal number of rounds for solving consensus is $f + 1$ [6], while the optimal number of rounds for solving set-agreement is $f/k + 1$ [10]. It is interesting whether this is a coincidence or an indication of an inherent connection between the two problems.

We presented a randomized algorithm for the at-most-once problem, which achieves optimal effectiveness. It is interesting to find algorithms that have a better step complexity, while retaining the

[2]We note that in [15] the number of processes is noted by m and the number of jobs by n, as opposed to this work.

optimal effectiveness. Our algorithm does not require knowledge of the number of jobs, and can even work for an online setting, but it still requires the number of processes to be known in advance, and it is an open question whether there is a solution which avoids this assumption.

We believe that similar algorithms to the ones presented in this paper can be constructed for weaker adversarial models. It is an open question whether there can be improved algorithms for weaker adversaries, and it is also important to find analogous algorithms for solving set-agreement in message-passing systems. Needless to say, obtaining lower bounds for these models is an important direction for further research.

Acknowledgements:. The author thanks Hagit Attiya for many valuable discussions, David Hay for a useful suggestion, and Noga Zewi for comments on an earlier version of this paper.

8. REFERENCES

[1] K. 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] J. Aspnes. Time- and space-efficient randomized consensus. *J. Algorithms*, 14(3):414–431, 1993.

[3] J. Aspnes and K. Censor. Approximate shared-memory counting despite a strong adversary. In *SODA '09: Proceedings of the Nineteenth Annual ACM -SIAM Symposium on Discrete Algorithms*, pages 441–450, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

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

[5] H. Attiya and K. Censor. Tight bounds for asynchronous randomized consensus. *J. ACM*, 55(5):1–26, 2008.

[6] H. Attiya and J. L. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2nd edition, 2004.

[7] E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 91–100, New York, NY, USA, 1993. ACM Press.

[8] G. Bracha and O. Rachman. Randomized consensus in expected $O(n^2 \log n)$ operations. In *Proceedings of the 5th International Workshop on Distributed Algorithms (WDAG)*, pages 143–150, 1991.

[9] S. Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993.

[10] S. Chaudhuri, M. Herlihy, N. A. Lynch, and M. R. Tuttle. Tight bounds for k-set agreement. *J. ACM*, 47(5):912–943, 2000.

[11] L. Cheung. Randomized wait-free consensus using an atomicity assumption. In *OPODIS*, pages 47–60, 2005.

[12] P. Ezhilchelvan, A. Mostefaoui, and M. Raynal. Randomized multivalued consensus. In *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Computing*, pages 195–200, 2001.

[13] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[14] M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, 1999.

[15] S. Kentros, A. Kiayias, N. C. Nicolaou, and A. A. Shvartsman. At-most-once semantics in asynchronous shared memory. In *DISC*, pages 258–273, 2009.

[16] A. Mostefaoui and M. Raynal. Randomized k-set agreement. In *SPAA '01: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 291–297, New York, NY, USA, 2001. ACM Press.

[17] A. Mostefaoui, M. Raynal, and F. Tronel. From binary consensus to multivalued consensus in asynchronous message-passing systems. *Inf. Process. Lett.*, 73(5-6):207–212, 2000.

[18] M. Raynal and C. Travers. Synchronous set agreement: a concise guided tour (including a new algorithm and a list of open problems). In *PRDC '06: Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, pages 267–274, Washington, DC, USA, 2006. IEEE Computer Society.

[19] M. Saks, N. Shavit, and H. Woll. Optimal time randomized consensus—making resilient algorithms fast in practice. In *Proceedings of the 2nd annual ACM-SIAM symposium on Discrete algorithms*, pages 351–362, 1991.

[20] M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM J. Comput.*, 29(5):1449–1483, 2000.

[21] G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[22] R. Turpin and B. A. Coan. Extending binary byzantine agreement to multivalued byzantine agreement. *Inf. Process. Lett.*, 18(2):73–76, 1984.

[23] J. Zhang and W. Chen. Bounded cost algorithms for multivalued consensus using binary consensus instances. *Inf. Process. Lett.*, 109(17):1005–1009, 2009.

APPENDIX

A. PROOF OF THE ITERATIVE $(\ell, k+1, n)$-AGREEMENT ALGORITHM

We now present the pseudocode of Algorithm 5 which solves $(\ell, k+1, n)$-agreement by iteratively decreasing the number of possible values using Algorithm 1, as discussed in Section 5.2.

Notice that Algorithm 1 is correct for agreeing on k values out of $k+1$ values, even if the $k+1$ possible input values are not necessarily $\{0, \ldots, k\}$, as long as they are a fixed and known set $\{v_0, \ldots, v_k\}$. This is done by having a bijective mapping between the two sets.

The following lemma guarantees the correctness of the algorithm.

LEMMA 15. *For each iteration s, $\ell \leq s \leq k$, the number of different values that appear in the $myValue$ variables of the processes that finish iteration s is at most s, and each of these values is the input of some process.*

PROOF. The proof is by induction over the iterations, where the base case is for $s = k$ and its proof is identical to that of Algorithm 1. For the induction step, we assume the lemma holds up to $s+1$ and prove it for s. A process finishes iteration s when it assigns $decide =$ true in line 13. This can only happen after it reads $myCheck[\text{disagree}] ==$ false in line 10, which implies that the

number of different entries in $myPropose$ that contain $true$ is at most s. Moreover, every value that is written to the $Propose[s]$ array is the $myValue$ variable of some process at the end of iteration $s+1$, and therefore is the input of some process, by the induction hypothesis. □

Algorithm 5 An $(\ell, k+1, n)$-agreement algorithm, code for process p_i

```
local variables: myValue, myPropose = [0..k],
   myCheck = [agree, disagree],s,m,r,decide
shared arrays: Propose[1..k][ ][0..k],
   Check[1..k][ ][agree, disagree]
1:  for s = k down to ℓ
2:    r = 1
3:    decide = false
4:    while decide == false
5:      Propose[s][r][myValue] = true
6:      myPropose = collect(Propose[s][r])
7:      if the number of entries in myPropose that contains true
          is at most s
8:        Check[s][r][agree] = ⟨true, myValue⟩
9:      else
10:       Check[s][r][disagree] = true
11:     myCheck = collect(Check[s][r])
12:     if myCheck[disagree] == false
13:       decide = true
14:     else if myCheck[agree] == ⟨true, v⟩
15:       myValue = v
16:     else if myCheck[agree] == false
17:       m = sharedCoin_{s+1}[r]
18:       myValue = the m-th entry in myPropose that
            contains true          // At most s + 1 such values
19:     r = r + 1
20:   end while
21: end for
22: return myValue
```

Applying Lemma 15 to $s = \ell$ gives the validity and agreement properties. This leads to the following theorem:

THEOREM 16. *Algorithm 5 solves $(\ell, k+1, n)$-agreement with $O(\sum_{s=k}^{\ell} \frac{I_{s+1}+k}{\delta_{s+1}})$ individual step complexity and $O(\sum_{s=k}^{\ell} \frac{T_{s+1}+nk}{\delta_{s+1}})$ total step complexity, where δ_{s+1}, I_{s+1}, and T_{s+1} are the agreement parameter, individual step complexity, and total step complexity, respectively, of the $(s+1)$-sided shared coins.*

PROOF. For each value of s, a process runs an iteration of the agreement algorithm for s out of $s+1$ values. By an analog of Theorem 1, this takes $O(\frac{I_{s+1}+k}{\delta_{s+1}})$ individual step complexity, and $O(\frac{T_{s+1}+nk}{\delta_{s+1}})$ individual step complexity. Notice that we add $O(k)$ steps for collecting the arrays and not $O(s)$ steps, since it may be that a process does not know which are the $s+1$ current possible values among the initial $k+1$ values.

Summing over all iterations gives the resulting complexities. □

Using the multi-sided shared coins of Section 4 gives:

Theorem 12 [restated] *Algorithm 5 solves $(\ell, k+1, n)$-agreement with $O((k-\ell+1)k + n(\log k - \log \ell))$ individual step complexity and $O((k-\ell+1)nk + n^2(\log k - \log \ell))$ total step complexity.*

PROOF. For the individual step complexity we have:

$$\begin{aligned}\sum_{s=k}^{\ell} \frac{I_{s+1}+k}{\delta_{s+1}} &= O(\sum_{s=k}^{\ell} \frac{n}{s} + k) \\ &= O((k-\ell+1)k + n\sum_{s=k}^{\ell} \frac{1}{s}) \\ &= O((k-\ell+1)k + n(\log k - \log \ell)),\end{aligned}$$

where the last equality follows from the fact that the harmonic series $H_k = \sum_{s=1}^{k} \frac{1}{s}$ is in the order of $\log k$. Similarly, we have that the total step complexity is $O((k-\ell+1)nk + n^2(\log k - \log \ell))$. □

Note that for $\ell = 1$, i.e., for agreeing on exactly one value out of the initial $k+1$ possible inputs, we get an individual step complexity of $O((k-\ell+1)k + n(\log k - \log \ell)) = O(k^2 + n \log k)$, and a total step complexity of $O((k-\ell+1)nk + n^2(\log k - \log \ell)) = O(nk^2 + n^2 \log k)$.

B. PROOF OF THE BIT-BY-BIT $(1, k+1, n)$-AGREEMENT ALGORITHM

We now present the pseudocode of Algorithm 6 which solves $(1, k+1, n)$-agreement by agreeing on every bit of the value, as discussed in Section 5.3.

Algorithm 6 A $(1, k+1, n)$-agreement algorithm by agreeing on $\log k$ bits, code for p_i

```
local variables: myValue = input, myPropose = [0.. log k],
   myCheck = [agree, disagree],r = 0,decide = false
shared arrays: Propose[1..k][ ][0.. log k],
   Check[1..k][ ][agree, disagree]
1:  for j = 1 ... ⌈log (k + 1)⌉
2:    while (decide == false)
3:      r+ = 1
4:      Propose[j][r][myValue[j]] = myValue
5:      myPropose = collect(Propose[j][r])
6:      if myPropose[0] ≠ ⊥ and myPropose[1] ≠ ⊥
7:        Check[j][r][disagree] = myPropose
8:      else Check[j][r][agree] = myValue
9:      myCheck = collect(Check[j][r])
10:     if myCheck[disagree] ≠ ⊥
11:       coin = sharedCoin_2(j, r)
12:       if myCheck[agree] ≠ ⊥
13:         myValue = Propose[j][r][myCheck[agree]]
14:       else myValue = myCheck[disagree][coin]
15:     else decide = true and r = 0
16:   end while
17: end for
18: return myValue
```

LEMMA 17. *For every j, $1 \le j \le \lceil \log (k+1) \rceil$, at the beginning of iteration j every process has $myValue$ that is the input of some process, and all the processes have $myValue$ with the same first $j-1$ bits.*

PROOF. The proof is by induction on j. The base case for $j = 1$ clearly holds since at the beginning of the algorithm $myValue$ is initialized to the input of the process, and $j - 1 = 0$ so there is no requirement from the first bits of $myValue$.

Induction step: Assume that the lemma holds up to value $j-1$. That is, the variable $myValue$ of all processes at the beginning of iteration $j-1$ has the same $j-2$ first bits, and they are all inputs of processes.

First, we notice that in iteration $j-1$ the variable $myValue$ can only change to a value written in the $Propose$ array in line 13, or to a value written in the $Check$ array in line 14. This implies that $myValue$ is always an input of some process.

Next, assume that at the end of the iteration processes p and q have $myValue$ variables with different first $j-1$ bits. By the induction hypothesis, this implies that their $j-1$-th bit is different. Let r be the first phase in which such two processes exist and decide in that phase. Assume, without loss of generality, that p executes line 4 after q does. This implies that when p reads the array $Propose$ in line 5, both entries are non-empty. But then p writes its value into the $disagree$ location of the array $Check$ and therefore cannot decide in that phase. □

Lemma 17, in an analog to Section 3, implies validity and agreement. The following theorem shows the correctness and complexity of the algorithm:

Theorem 13 [restated] *Algorithm 6 solves $(1, k+1, n)$-agreement with $O(\lceil \log(k+1) \rceil \cdot \frac{I}{\delta})$ individual step complexity and $O(\lceil \log(k+1) \rceil \frac{T}{\delta})$ total step complexity.*

PROOF. In each iteration j, $1 \leq j \leq \lceil \log(k+1) \rceil$, by an analog to Lemma 4, the expected number of phases until all non-faulty process decide is $1 + 1/\delta$ which is $O(\frac{1}{\delta})$. In each phase, a process takes $O(1)$ steps in addition to the I steps it takes in the sharedCoin$_2$ algorithm. Therefore, the individual step complexity of Algorithm 6 is $O(\lceil \log(k+1) \rceil \cdot \frac{I}{\delta})$, and the total step complexity is $O(\lceil \log(k+1) \rceil \frac{T}{\delta})$. □