Communication-Efficient Randomized Consensus

Dan Alistarh[1], James Aspnes[2], Valerie King[3], and Jared Saia[4]

[1] Microsoft Research, Cambridge, UK. Email: dan.alistarh@microsoft.com
[2] Yale University, Department of Computer Science. Email: aspnes@cs.yale.edu.
[3] University of Victoria; Simons Institute for the Theory of Computing, Berkeley; Institute for Advanced Study, Princeton. Email: val@cs.uvic.ca
[4] University of New Mexico, Department of Computer Science. Email: saia@cs.unm.edu

Abstract. We consider the problem of consensus in the challenging *classic* model. In this model, the adversary is adaptive; it can choose which processors crash at any point during the course of the algorithm. Further, communication is via asynchronous message passing: there is no known upper bound on the time to send a message from one processor to another, and all messages and coin flips are seen by the adversary.

We describe a new randomized consensus protocol with expected message complexity $O(n^2 \log^2 n)$ when fewer than $n/2$ processes may fail by crashing. This is an almost-linear improvement over the best previously known protocol, and within logarithmic factors of a known $\Omega(n^2)$ message lower bound. The protocol further ensures that no process sends more than $O(n \log^3 n)$ messages in expectation, which is again within logarithmic factors of optimal. We also present a generalization of the algorithm to an arbitrary number of failures t, which uses expected $O(nt + t^2 \log^2 t)$ total messages. Our protocol uses messages of size $O(\log n)$, and can therefore scale to large networks.

Our approach is to build a message-efficient, resilient mechanism for aggregating individual processor votes, implementing the message-passing equivalent of a weak shared coin. Roughly, in our protocol, a processor first announces its votes to small groups, then propagates them to increasingly larger groups as it generates more and more votes. To bound the number of messages that an individual process might have to send or receive, the protocol progressively increases the weight of generated votes. The main technical challenge is bounding the impact of votes that are still "in flight" (generated, but not fully propagated) on the final outcome of the shared coin, especially since such votes might have different weights. We achieve this by leveraging the structure of the algorithm, and a technical argument based on martingale concentration bounds. Overall, we show that it is possible to build an efficient message-passing implementation of a shared coin, and in the process (almost-optimally) solve the classic consensus problem in the asynchronous message-passing model.

1 Introduction

Consensus [28, 29] is arguably the most well-studied problem in distributed computing. The FLP impossibility result [21], showing that consensus could not be achieved deterministically in an asynchronous message-passing system with even one crash failure, sparked a flurry of research on overcoming this fundamental limitation, either by adding timing assumptions, e.g. [19], employing failure detectors, e.g. [17], or by relaxing progress conditions to allow for *randomization*, e.g. [13]. A significant amount

of research went into isolating time and space complexity bounds for randomized consensus in the shared-memory model, e.g. [4, 5, 9, 10, 12, 15, 20], developing elegant and technically complex tools in the process. As a result, the time complexity of consensus in asynchronous shared memory is now well characterized: the tight bound on total number of steps is $\Theta(n^2)$ [12], while the individual step bound is $\tilde{\Theta}(n)$ [5].[5]

Somewhat surprisingly, the complexity of randomized consensus in the other core model of distributed computing, the *asynchronous message-passing model*, is much less well understood. In this model, communication is via full-information, asynchronous message passing: there is no known upper bound on the time to send a message from one processor to another, and all messages are seen by the adversary. Further, as in the shared memory model, the adversary is adaptive; it can choose which processors crash at any point during the course of the algorithm. We refer to this as the *classic* model.

While simulations exist [11] allowing shared-memory algorithms to be translated to message-passing, their overhead is at least linear in the number of nodes. It is therefore natural to ask if message-efficient solutions for randomized consensus can be achieved, and in particular if quadratic shared-memory communication cost for consensus can be also achieved in message-passing systems against a strong, adaptive adversary.

In this paper, we propose a new randomized consensus protocol with expected message complexity $O(n^2 \log^2 n)$ against a strong (adaptive) adversary, in an asynchronous message-passing model in which less than $n/2$ processes may fail by crashing. This is an almost-linear improvement over the best previously known protocol. Our protocol is also *locally-efficient*, ensuring that no process sends or receives more than expected $O(n \log^3 n)$ messages, which is within logarithmic factors of the linear lower bound [12]. We also provide a generalization to an arbitrary number of failures $t < n/2$, which uses $O(nt + t^2 \log^2 t)$ messages.

Our general strategy is to construct a **message-efficient weak shared coin**. A weak shared coin with parameter $\delta > 0$ is a protocol in which for each possible return value ± 1, there is a probability of at least δ that all processes return that value. We then show that this shared coin can be used in a message-efficient consensus protocol modeled on a classic shared-memory protocol of Chandra [16].

Since early work by Bracha and Rachman [15], implementations of weak shared coins for shared-memory systems with an adaptive adversary have generally been based on voting. If the processes between them generate n^2 votes of ± 1, then the absolute value of the sum of these votes will be at least n with constant probability. If this event occurs, then even if the adversary hides $\Theta(n)$ votes by crashing processes, the total vote seen by the survivors will still have the same sign as the actual total vote.

Such algorithms can be translated to a message-passing setting directly using the classic Attiya-Bar-Noy-Dolev (ABD) simulation [11]. The main idea of the simulation is that a write operation to a register is simulated by distributing a value to a majority of the processes (this is possible because of the assumption that a majority of the processes do not fail). Any subsequent read operation contacts a majority of the processes, and because the majorities overlap, this guarantees that any read sees the value of previous writes.

The obvious problem with this approach is that its message complexity is high: because ABD uses $\Theta(n)$ messages to implement a write operation, and because each

[5] We consider a model with n processes, $t < n/2$ of which may fail by crashing. The $\tilde{\Theta}$ notation hides logarithmic factors.

vote must be written before the next vote is generated if we are to guarantee that only $O(n)$ votes are lost, the cost of this direct translation is $\Theta(n^3)$ messages. Therefore, the question is whether this overhead can be eliminated.

Our approach. To reduce both total and local message complexity, we employ two new ingredients.

The first is an algorithmic technique to reduce the message complexity of distributed vote counting by using a binary tree of process groups called **cohorts**, where each leaf corresponds to a process, and each internal node represents a cohort consisting of all processes in the subtree. Instead of announcing each new vote to all participants, new ± 1 votes are initially only announced to small cohorts at the bottom of the tree, but are propagated to increasingly large cohorts as more votes are generated. As the number of votes grows larger, the adversary must crash more and more processes to hide them. This generalizes the one-crash-one-vote guarantee used in shared-memory algorithms to a many-crashes-many-votes approach.

At the same time, this technique renders the algorithm message-efficient. Given a set of generated votes, the delayed propagation scheme ensures that each vote accounts for exactly one update at the leaf, $1/2$ updates (amortized) at the 2-neighborhood, and in general, $1/2^i$ (amortized) updates at the ith level of the tree. Practically, since the ith level cohort has 2^i members, the propagation cost of a vote is exactly one message per tree level. In total, that is $\log n$ messages per vote, amortized.

A limitation of the above scheme is that a fast process might have to generate all the $\Theta(n^2)$ votes itself in order to decide, which would lead to high individual message complexity. The second technical ingredient of our paper is a procedure for assigning increasing weight to a processes' votes, which reduces individual complexity. This general idea has previously been used to reduce individual work for shared-memory randomized consensus [5, 7, 10]; however, we design and analyze a new weighting scheme that is customized for our vote-propagation mechanism.

In our scheme, each process doubles the weight of its votes every $4n \log n$ votes, and we run the protocol until the total reported variance—the sum of the squares of the weights of all reported votes—exceeds $n^2 \log n$. Intuitively, this allows a process running alone to reach the threshold quickly, reducing per-process message complexity. This significantly complicates the termination argument, since a large number of generated votes, of various weights, could be still making their way to the root at the time when a process first notices the termination condition. We show that, with constant probability, this drift is not enough to influence the sign of the sum, by carefully bounding the weight of the extra votes via the structure of the algorithm and martingale concentration bounds. We thus obtain a constant-bias weak shared coin. The bounds on message complexity follow from bounds on the number of votes generated by any single process or by all the processes together before the variance threshold is reached.

We convert the shared coin construction into a consensus algorithm via a simple framework inspired by Chandra's shared-memory consensus protocol [16], which in turn uses ideas from earlier consensus protocols of Chor, Israeli, and Li [18] and Aspnes and Herlihy [9]. Roughly, we associate each of the two possible decision values with a message-passing implementation of a max register [6, 8], whose value is incremented by the "team" of processes obtaining that value from the shared coin. If a process sees that its own team has fallen behind, it switches to the other team, and once one of the max register's values surpasses the other by two, the corresponding team wins. Ties

are broken (eventually) by having processes that do not observe a clear leader execute a weak shared coin. This simple protocol maintains the asymptotic complexity of the shared coin in expectation.

Finally, we present a more efficient variant of the protocol for the case where $t = o(n)$, based on the observation that we can "deputize" a subset of $2t+1$ of the processes to run the consensus protocol, and broadcast their result to all n processes. The resulting protocol has total message complexity $O(nt + t^2 \log^2 t)$, and $O(n + t \log^3 t)$ individual message complexity.

Overall, we show that it is possible to build message-efficient weak shared coins and consensus in asynchronous message-passing systems. Our vote counting construction implements a message-efficient, asynchronous approximate trigger counter [25], which may be of independent interest. An interesting aspect of our constructions is that message sizes are small: since processes only communicate vote counts, messages only require $O(\log n)$ bits of communication.

2 System Model and Problem Statement

We consider the standard asynchronous message-passing model, in which n processes communicate with each other by sending messages through channels. We assume that there are two uni-directional channels between any pair of processes. Communication is *asynchronous*, in that messages can be arbitrarily delayed by a channel, and in particular may be delivered in arbitrary order. However, we assume that messages are not corrupted by the channel.

Computation proceeds in a sequence of *steps*. At each step, a process checks incoming channels for new messages, then performs local computation, and sends new messages. A process may become *faulty*, in which case it ceases to perform local computation and to send new messages. A process is *correct* if it takes steps infinitely often during the execution. We assume that at most $t < n/2$ processes may be faulty during the execution.

Message delivery and process faults are assumed to be controlled by a *strong (adaptive)* adversary. At any time during the computation, the adversary can examine the entire state of the system (in particular, the results of process coinflips), and decide on process faults and messages to be delivered.

The *(worst-case) message complexity* of an algorithm is simply the maximum, over all adversarial strategies, of the total number of messages sent by processes running the algorithm. Without loss of generality, we assume that the adversary's goal is to maximize the message complexity of our algorithm.

In the *(binary) randomized consensus* problem, each process starts with an input in $\{0, 1\}$, and returns a decision in $\{0, 1\}$. A correct protocol satisfies *agreement*: all processes that return from the protocol choose the same decision, *validity*: the decision must equal some process's input, and *probabilistic termination*: every non-faulty process returns after a finite number of steps, with probability 1.

3 Related Work

The first shared-memory protocol for consensus was given by Chor, Israeli, and Li [18] for a weak adversary model, and is based on a race between processors to impose their

proposals. Abrahamson [1] gave the first wait-free consensus protocol for a strong adversary, taking exponential time. Aspnes and Herlihy [9] gave the first polynomial-time protocol, which terminates in $O(n^4)$ expected total steps. Subsequent work, e.g. [4, 10, 15, 30], continued to improve upper and lower bounds for this problem, until Attiya and Censor [12] showed a tight $\Theta(n^2)$ bound on the total number of steps for asynchronous randomized consensus. In particular, their lower bound technique implies an $\Omega(t(n-t))$ total message complexity lower bound and a $\Omega(t)$ individual message complexity lower bound for consensus in the asynchronous message-passing model. Our $(n/2-1)$-resilient algorithms match both lower bounds within logarithmic factors, while the t-resilient variant matches the first lower bound within logarithmic factors.

To our knowledge, the best previously known upper bound for consensus in asynchronous message-passing requires $\Theta(n^3)$ messages. This is obtained by simulating the elegant shared-memory protocol of Attiya and Censor-Hillel [12], using the simulation from [11]. A similar bound can be obtained by applying the same simulation to an $O(n)$-individual-work algorithm of Aspnes and Censor [7].

In the message passing model, significant work has focused on the problem of Byzantine agreement, which is identical to consensus except that the adversary *controls* up to t processes, and can cause them to deviate arbitrarily from the protocol. In 1983, Fischer, Lynch and Patterson [21] showed that no deterministic algorithm could solve consensus, and hence Byzantine agreement, in the classic model. In the same year, Ben-Or gave a randomized algorithm for Byzantine agreement which required an expected exponential communication rounds and number of messages [13]. His algorithm tolerated $t < n/5$. Subsequent work extended this idea in two directions: to solve message-passing Byzantine agreement faster and with higher resilience, and to solve agreement wait-free, in asynchronous shared-memory tolerating crash failures.

Resilience against Byzantine faults was improved to $t < n/3$ in 1984 by Bracha [14]. However, the communication rounds and number of messages remained exponential in expectation. This resilience is the best possible for randomized Byzantine agreement [24]. In 2013, King and Saia gave the first algorithm for Byzantine agreement in the classic model with expected polynomial communication rounds and number of messages [26]. Their algorithm required in expectation $O(n^{2.5})$ communication rounds, $O(n^{6.5})$ messages, and $O(n^{7.5})$ bits sent. It tolerated $t < n/500$. Unfortunately, local computation time was exponential. In 2014, the same authors achieved polynomial computation time. However, the new algorithm required expected $O(n^3)$ communication rounds, $O(n^7)$ messages, and $O(n^8)$ bits sent. Further, the resilience decreased to $t < 0.000028n$ [27].

4 A Message-Passing Max Register

To coordinate the recording of votes within a group, we use a message-passing max register [6]. The algorithm is adapted from [8], and is in turn based on the classic ABD implementation of a message-passing register [11]. The main change from [8] is that we allow for groups consisting of $g < n$ processes. Recall that a max register maintains a value v, which is read using the *MaxRead* operation, and updated using the *MaxUpdate* operation. A $MaxUpdate(u)$ operation changes the value only if u is higher than the current value v.

Description. We consider a group G of g processes, which implement the max register R collectively. Each process p_i in the group maintains a current value estimate v_i locally. The communicate procedure [11] broadcasts a request to all processes in the group G, and waits for at least $\lceil g/2 \rceil$ replies.[6]

To perform a MaxRead, the process communicates a $MaxRead(R)$ request to all other processes, setting its value v_i to be the maximum value received. Before returning this value, the process communicates a $MaxReadACK(R, v_i)$ message. All processes receiving such a message will update their current estimate of R, if this value was less than v_i. If it receives at least $\lceil g/2 \rceil$ replies, the caller returns v_i as the value read. This ensures that, if a process p_i returns v_i, no other process may later return a smaller value for R.

A MaxUpdate with input u is similar to a MaxRead: the process first communicates a $MaxUpdate(R, u)$ message to the group, and waits for at least $\lceil g/2 \rceil$ replies. Process p_i sets its estimate v_i to the maximum between u and the maximum value received in the first round, before communicating this value once more in a second broadcast round. Again, all processes receiving this message will update their current estimate of R, if necessary. The algorithm ensures the following properties.

Lemma 1. *The max register algorithm above implements a linearizable max register. If the communicate procedure broadcasts to a group G of processes of size g, then the message complexity of each operation is $O(g)$, and the operation succeeds if at most $\lfloor g/2 \rfloor$ processes in the group are faulty.*

5 The Weak Shared Coin Algorithm

We now build a message-efficient asynchronous weak shared coin. Processes generate random votes, whose weight increases over time, and progressively communicate them to groups of nodes of increasing size. This implements a shared coin with constant bias, which in turn can be used to implement consensus.

Vote Propagation. The key ingredient is a message-efficient construction of an approximate asynchronous vote counter, which allows processes to maintain an estimate of the total number of votes generated, and of their sum and variance. The distributed vote counter is structured as a binary tree, where each process is associated with a leaf. Each subtree of height h is associated with a *cohort* of 2^h processes, corresponding to its leaves. To each such subtree s, we associate a max register R_s, implemented as described above, whose value is maintained by all the processes in the corresponding cohort. For example, the value at each leaf is only maintained by the associated process, while the root value is tracked by all processes.

The max register R_s corresponding to the subtree rooted at s maintains three values: the *count*, an estimate of the number of votes generated in the subtree, *total*, an estimate of the sum of generated votes, and *var*, an estimate of the variance of the generated votes. Values are ordered only by the first component. Practically, the implementation

[6] Since $t < n/2$ processes may crash, and g may be small, a process may block while waiting for replies. This only affects the progress of the protocol, but not its safety. Our shared coin implementation will partition the n processes into max register groups, with the guarantee that some groups always make progress.

```
1  Let K = n^2 log_2 n
2  Let T = 4n log_2 n
3  count ← 0
4  var ← 0
5  total ← 0
6  for k ← 1, 2, ..., ∞ do
7      Let w_k = 2^⌊(k−1)/T⌋
8      Let vote = ±w_k with equal probability
9      count ← count + 1
10     var ← var + w_k^2
11     total ← total + vote
12     Write ⟨count, var, total⟩ to max register for my leaf
13     for j ← 1 ... log_2 n do
14         if 2^j does not divide k then
15             break
16         Let s be my level-j ancestor, with children s_ℓ and s_r
17         in parallel do
               /* read left and right counts                    */
18             ⟨count_ℓ, var_ℓ, total_ℓ⟩ ← ReadMax(s_ℓ)
19             ⟨count_r, var_r, total_r⟩ ← ReadMax(s_r)
           /* update the parent                                 */
20         WriteMax(s, ⟨count_ℓ + count_r, var_ℓ + var_r, total_ℓ + total_r⟩)
21     if n divides k then
22         ⟨count_root, var_root, total_root⟩ ← ReadMax(root)
           /* if the root variance exceeds the threshold        */
23         if var_root ≥ K then
24             return sgn(total_root)        /* return sign of root total */
```

Algorithm 1: Shared coin using increasing votes.

is identical to the max register described in the previous section, except that whenever sending the *count*, processes also send the associated *total* and *var*. Processes always adopt the tuple of maximum *count*. If a process receives two tuples with the same *count* but different *total*/*var* components, they adopt the one with the maximum total.

A process maintains max register estimates for each subtree it is part of. Please see Algorithm 1 for the pseudocode. In the kth iteration of the shared coin, the process generates a new vote with weight $\pm w_k$ chosen as described in the next paragraph. After generating the vote, the process will propagate its current set of votes up to level r, the highest power of two which divides k (line 15). At each level from 1 (the leaf's parent) up to r, the process reads the max registers left and right children, and updates the $\langle count, total, var\rangle$ of the parent to be the sum of the corresponding values at the child max registers (lines 17–20).

If n divides k, then the process also checks the count at the root. If the root variance count is greater than the threshold of K votes, the process returns the sign of the root total as its output from the shared coin. Otherwise, the process continues to generate votes.

Vote Generation. Each process generates votes with values $\pm w_k$ in a series of epochs, each epoch consisting of $T = 4n \log_2 n$ loop iterations. Within each epoch, all votes have the same weight, and votes are propagated up the tree of max registers using the schedule described above. At the start of a new epoch, the weight of the votes doubles. This ensures that only $O(\log n)$ epochs are needed until a single process can generate enough variance by itself to overcome the offsets between the observed vote and the generated vote due to delays in propagation up the tree.

Because votes have differing weights, we track the total variance of all votes included in a max register in addition to their number, and continue generating votes until this total variance exceeds a threshold $K = n^2 \log_2 n$, at which point the process returns the sign of the root total (line 24).

6 Algorithm Analysis

We prove that the algorithm in Section 5 implements a correct weak shared coin. We first analyze some of the properties of the tree-based vote counting structure. For simplicity, we assume that the number of processes n is a power of two. Due to space constraints, the complete argument is given in the full version of the paper [3].

Vote Propagation. The algorithm is based on the idea that, as processes take steps, counter values for the cohorts get increased, until, eventually, the root counter value surpasses the threshold, and processes start to return. We first provide a way of associating a set of generated votes to each counter value.

We say that a process p_i counts a number x_i of (consecutive) locally-generated votes to node s if, after generating the last such vote, process p_i updates the max register at s in its loop iteration. We prove that this procedure has the following property:

Lemma 2. *Consider a subtree rooted at node s with ℓ leaves, corresponding to member processes $q_1, q_2, \ldots, q_\ell$. Let $x_1, x_2, \ldots, x_\ell$ be the number of votes most recently counted by processes $q_1, q_2, \ldots, q_\ell$ at node s, respectively. Then the value of the* count *component of the max register at s is at least $\sum_{m=1}^{\ell} x_m$.*

Proof Strategy. The proof can be divided into three steps. The first shows that the coin construction offers a good approximation of the generated votes, i.e. the total vote U^t observed in the root max register at any time t is close to the actual total generated vote V^t at the same time. The second step shows that when the threshold K is crossed at some time t, the **common votes total** $|V^t|$ is likely to be far away from 0. The last step shows that, for any subsequent time t', the combination of the approximation slack $U^{t'} - V^{t'}$ and any **extra votes** $V^{t'} - V^t$ observed by a particular process at time t' will not change the sign of the total vote.

The first step involves a detailed analysis of what votes may be omitted from the visible total combined with an extension of the Azuma-Hoeffding inequality [10]; the second step requires use of a martingale Central Limit Theorem [23, Theorem 3.2]; the last follows from an application of Kolmogorov's inequality. (For background on martingales, we point the reader to [22].) We begin by stating a few technical claims.

Lemma 3. *Fix some execution of Algorithm 1, let n_i be the number of votes generated by process p_i during this execution, and let w_{n_i} be the weight of the n_i-th vote. Then*

1. $\sum_{i=1}^{n}\sum_{j=1}^{n_i} w_j^2 \le \frac{K+2n^2}{1-8n/T} = O(n^2 \log n)$.
2. $w_{n_i} \le \sqrt{1+\frac{4K+8n^2}{T-8n}} = O(\sqrt{n})$.
3. *For all* j, $n_j = O(n \log^2 n)$.
4. $\sum_i w_{n_i}^2 \le n + \frac{4K+8n^2}{T-8n} = O(n)$.
5. $\sum_i n_i = O(n^2 \log n)$.

For any adversary strategy $\mathcal{A}$, let $\tau_{\mathcal{A}}$ be a stopping time corresponding to the first time t at which $U^{\mathsf{root}}[t].\mathsf{var} \ge K$. We will use a martingale Central Limit Theorem to show that $V^{\mathsf{root}}[\tau_{\mathcal{A}}].\mathsf{total}$ converges to a normal distribution as n grows, when suitably scaled. This will then be used to show that all processes observe a population of common votes whose total is likely to be far from zero. The notation $X \xrightarrow{p} Y$ means that X converges in probability to Y, and $Y \xrightarrow{d} Y$ means that X converges in distribution to Y. We show that the following convergence holds.

Lemma 4. *Let* $\{\mathcal{A}_n\}$ *be a family of adversary strategies, one for each number of processes* $n \in \mathbb{N}$. *Let* $\tau_n = \tau_{\mathcal{A}_n}$ *be as above. Then*

$$\frac{V^{\mathsf{root}}[\tau_n].\mathsf{total}}{\sqrt{K}} \xrightarrow{d} N(0,1). \tag{1}$$

Once this is established, for each subtree s and time t_s, let $D^s[t_s] = V^s[t_s].\mathsf{total} - U^s[t_s].\mathsf{total}$ be the difference between the generated votes in s at time t_s and the votes reported to the max register corresponding to s at time t_s. Let s_ℓ and s_r be the left and right subtrees of s, and let t_{s_ℓ} and t_{s_r} be the times at which the values added to produce $U^s[t_s].\mathsf{total}$ were read from these subtrees. Recursing over all proper subtrees of s, we obtain that

$$D^s[t_s] = V^s[t_s].\mathsf{total} - U^s[t_s].\mathsf{total} = \sum_{s'} \left(V^{s'}[t_{\text{parent}(s')}].\mathsf{total} - V^{s'}[t_{s'}].\mathsf{total} \right),$$

where s' ranges over all proper subtrees of s.

To bound this sum, we consider each (horizontal) layer of the tree separately, and observe that the missing interval of votes $\left(V^{s'}[t_{\text{parent}(s')}].\mathsf{total} - V^{s'}[t_{s'}].\mathsf{total} \right)$ for each subtree s in layer h consists of at most 2^h votes by each of at most 2^h processes. For each process p_i individually, the variance of its 2^h heaviest votes, using Lemma 3, is at most $2^h \left(1 + (4/T)\sum_{j=1}^{n_i} w_j^2\right)$. If we sum the total variance of at most 2^h votes from all processes, we get at most

$$2^h \left(n^2 + (4/T)\sum_{i=1}^{n}\sum_{j=1}^{n_i} w_j^2 \right) \le 2^h \left(n^2 + \frac{K+2n^2}{1-8n/T} \right),$$

again using Lemma 3.

We would like to use this bound on the total variance across all missing intervals to show that the sum of the total votes across all missing intervals is not too large. Intuitively, if we can apply a bound to the total variance on a particular interval, we

expect the Azuma-Hoeffding inequality to do this for us. But there is a complication in that the total variance for an interval may depend in a complicated way on the actions taken by the adversary during the interval. So instead, we attack the problem indirectly, by adopting a different characterization of the relevant intervals of votes and letting the adversary choose between them to obtain the actual intervals that contributed to $D^{\mathsf{root}}[t]$. We will use the following extended version of the classic Azuma-Hoeffding inequality [10]:

Lemma 5 ([10, Theorem 4.5]). *Let $\{S_0, \mathcal{F}_0\}$, $0 \leq i \leq n$ be a zero-mean martingale with difference sequence $\{X_i\}$. Let w_i be measurable $\mathcal{F}_{i-1}$, and suppose that for all i, $|X_i| \leq w_i$ with probability 1; and that there exists a bound W such that $\sum_{i=1}^{n} w_i^2 \leq W$ with probability 1. Then for any $\lambda > 0$,*

$$\Pr\left[S_n \geq \lambda\right] \leq e^{-\lambda^2/2W}. \tag{2}$$

Fix an adversary strategy. For each subtree s, let $X_1^s, X_2^s, \ldots$ be the sequence of votes generated in s. For each s, t, and W, let $Y_i^{tsW} = X_i^s$ if (a) at least t votes have been generated by all processes before X_i^s is generated, *and* (b) $\sum_{j<i}(Y_i^{tsW})^2 + (X_i^s)^2 \leq W$. Otherwise, let Y_i^{tsW} be 0. If we let $\mathcal{F}_i$ be generated by all votes preceding X_i^s, then the events (a) and (b) are measurable $\mathcal{F}_i$, so $\{Y_i^{tsW}, \mathcal{F}_i\}$ forms a martingale. Furthermore, since only the sign of Y_i^{tsW} is unpredictable, we can define $w_i = (Y_i^{tsW})^2$ to fit Lemma 5. From (b), we have that $\sum w_i^2 \leq W$ always. It follows that, for any $c > 0$,

$$\Pr\left[\sum_i Y_i^{tsW} \geq \sqrt{2cW \ln n}\right] \leq e^{-c \ln n} = n^{-c}.$$

There are polynomially many choices for the parameters t, s, and W. Union bounding over all such choices shows that, for c sufficiently large, with high probability $\sum_i Y_i^{tsW}$ is bounded by $\sqrt{2cW \ln n}$ for all such intervals. We now use this to show the following.

Lemma 6. *For any adversary strategy and sufficiently large n, with probability $1 - o(1)$, it holds that at all times t,*

$$\left|V^{\mathsf{root}}[t].\mathsf{total} - U^{\mathsf{root}}[t].\mathsf{total}\right| \leq 6n\sqrt{\log_2 n}.$$

Proof. We are trying to bound $D^{\mathsf{root}}[t] = \sum_s (V^s[t_{\text{parent}(s)}] - V^s[t_s])$, where s ranges over all proper subtrees of the tree and for each s of size 2^h, the interval $(t_s, t_{\text{parent}(s)}]$ includes at most 2^h votes for each process.

Suppose that for each t, s, W, it holds that $Y^{tsW} \leq \sqrt{9W \ln n}$. By the preceding argument, each such event fails with probability at most $n^{-9/2}$. There are $O(n^2 \log n)$ choices for t, $O(n)$ choices for s, and $O(n^2 \log n)$ choices for W, so taking a union bound over all choices of Y^{tsW} not occurring shows that this event occurs with probability $O(n^{-1/2} \log^2 n) = o(1)$.

If all Y^{tsW} are bounded, then it holds deterministically that

$$\sum_s (V^s[t_{\text{parent}(s)}] - V^s[t_s]) = \sum_{h=0}^{\log_2 n-1} \sum_{s,|s|=2^h} (V^s[t_{\text{parent}(s)}] - V^s[t_s])$$
$$= \sum_{h=0}^{\log_2 n-1} \sum_{s,|s|=2^h} Y^{t_s s W_s} \leq \sum_{h=0}^{\log_2 n-1} \sum_{s,|s|=2^h} \sqrt{2cW_s \ln n}$$
$$= \sqrt{2c \ln n} \sum_{h=0}^{\log_2 n-1} \sum_{s,|s|=2^h} \sqrt{W_s} \leq \sqrt{9 \ln n} \sum_{h=0}^{\log_2 n-1} \sum_{s,|s|=2^h} \sqrt{W_s},$$

where W_s is the total variance of the votes generated by s in the interval $(t_s, t_{\text{parent}(s)}]$.

Note that this inequality does not depend on analyzing the interaction between voting and when processes read and write the max registers. For the purposes of computing the total offset we are effectively allowing the adversary to choose what intervals it includes retrospectively, after carrying out whatever strategy it likes for maximizing the probability that any particular values Y^{tsW} are too big.

Because each process i in s generates at most 2^h votes, and each such vote has variance at most $w_{n_i}^2$, we have

$$W_s \leq 2^h \sum_{i \in s} w_{n_i}^2.$$

Furthermore, the subtrees at any fixed level h partition the set of processes, so applying Lemma 3 gives

$$\sum_{s,|s|=2^h} W_s \leq \sum_{s,|s|=2^h} 2^h \sum_{i \in s} w_{n_i}^2 = 2^h \sum_i w_{n_i}^2 \leq 2^h \left(n + \frac{4K + 8n^2}{T - 8n} \right).$$

By concavity of square root, $\sum \sqrt{x_i}$ is maximized for non-negative x_i constrained by a fixed bound on $\sum \sqrt{x_i}$ by setting all x_i equal. Setting all $n/2^h$ values W_s equal gives the following upper bound.

$$W_s \leq \frac{2^h}{n} \cdot 2^h \left(n + \frac{4K + 8n^2}{T - 8n} \right) = 2^{2h} \left(1 + \frac{4K + 8n^2}{Tn - 8n^2} \right), \text{ and thus}$$
$$\sqrt{W_s} \leq 2^h \sqrt{1 + \frac{4K + 8n^2}{Tn - 8n^2}},$$

which gives the bound

$$\sqrt{9\ln n}\sum_{h=0}^{\log_2 n-1}\sum_{s,|s|=2^h}\sqrt{W_s} \le \sqrt{9\ln n}\sum_{h=0}^{\log_2 n-1}2^h\sqrt{1+\frac{4K+8n^2}{Tn-8n^2}}$$

$$= \sqrt{9\ln n\left(1+\frac{4K+8n^2}{Tn-8n^2}\right)}\sum_{h=0}^{\log_2 n-1}2^h = 3(n-1)\sqrt{\ln n}\sqrt{1+\frac{4K+8n^2}{Tn-8n^2}}$$

$$= 3(n-1)\sqrt{\ln n}\sqrt{1+\frac{\log_2 n+2}{\log_2 n-2}} \le 6n\sqrt{\log_2 n},$$

when n is sufficiently large. The last step uses the fact that for $K = n^2\log n$ and $T = 4n\log n$, the value under the radical converges to 2 in the limit, and $3\sqrt{2/\ln 2} < 6$.

For the last step of the proof, we need to show that the **extra votes** that arrive after K variance has been accumulated are not enough to push $V^{\mathsf{root}}[t]$ close to the origin. For this, we use Kolmogorov's inequality, a martingale analogue to Chebyshev's inequality, which says that if we are given a zero-mean martingale $\{S_i, \mathcal{F}_i\}$ with bounded variance, then $\Pr[\exists i \le n : |S_i| \ge \lambda] \le \frac{\lambda^2}{\mathrm{Var}[S_n]}$.

Consider the martingale $S_1, S_2, \ldots$ where S_i is the sum of the first i votes after $V^{\mathsf{root}}[i].\mathsf{var}$ first passes K. Then from Lemma 3,

$$\mathrm{Var}[S_i] \le \frac{K+2n^2}{1-8n/T} - K = \frac{8K(n/T)+2n^2}{1-8n/T} = \frac{8n^2+2n^2}{1-2/(\log_2 n)} = O(n^2).$$

So for any fixed c, the probability that $|S_i|$ exceeds cK for any i is $O(1/\log n) = o(1)$.

Final argument. From Lemma 4, we have that the total common vote $V^{\mathsf{root}}[\tau_n].\mathsf{total}$ converges in distribution to $N(0,1)$ when scaled by $\sqrt{K} = n\sqrt{\log_2 n}$. In particular, for any fixed constant c, there is a constant probability $\pi_c > 0$ that for sufficiently large n, $\Pr\left[V^{\mathsf{root}}[\tau_n] \ge cn\sqrt{\log_2 n}\right] \ge \pi_c$.

Let c be 7. Then with probability $\pi_7 - o(1)$, all of the following events occur:

1. The common vote $V^{\mathsf{root}}[\tau_n].\mathsf{total}$ exceeds $7n\sqrt{\log_2 n}$;
2. For any i, the next i votes have sum $o(n\sqrt{\log_2 n})$;
3. The vote $U^{\mathsf{root}}[t].\mathsf{total}$ observed by any process differs from $V^{\mathsf{root}}[t].\mathsf{total}$ by at most $6n\sqrt{\log_2 n}$.

If this occurs, then every process observes, for some t, $U^{\mathsf{root}}[t].\mathsf{total} \ge 7n\sqrt{\log_2 n} - 6n\sqrt{\log_2 n} - o(n\sqrt{\log n}) > 0$. In other words, all processes return the same value $+1$ with constant probability for sufficiently large n. By symmetry, the same is true for -1. We have therefore constructed a weak shared coin with constant agreement probability.

Theorem 1. *Algorithm 1 implements a weak shared coin with constant bias, message complexity $O(n^2\log^2 n)$, and with a bound of $O(n\log^3 n)$ on the number of messages sent and received by any one process.*

Proof. We have just shown that Algorithm 1 implements a weak shared coin with constant bias, and from Lemma 3 we know that the maximum number of votes generated by any single process is $O(n \log^2 n)$. Because each process communicates with a subtree of 2^h other processes every 2^{-h} votes, each level of the tree contributes $\Theta(1)$ amortized outgoing messages and incoming responses per vote, for a total of $\Theta(\log n)$ messages per vote, or $O(n \log^3 n)$ messages altogether.

In addition, we must count messages received and sent by a process p as part of the max register implementation. Here for each process q in p's level-h subtree, p may incur $O(1)$ messages every 2^h votes generated by q. Each such process q generates at most $O(n \log^2 n)$ votes, and there are 2^h such processes q. So p incurs a total of $O(n \log^2 n)$ votes from its level-h subtree. Summing over all $\log n$ levels gives the same $O(n \log^3 n)$ bound on messages as for max-register operations initiated by p.

This gives the final bound of $O(n \log^3 n)$ messages per process. Applying the same reasoning to the total vote bound from Lemma 3 yields the bound of $O(n^2 \log^2 n)$ on total message complexity.

7 Consensus Protocol and Extension for General t

Consensus. We now describe how to convert a message-efficient weak shared coin into message-efficient consensus. We adapt a shared-memory consensus protocol, due to Chandra [16], which, like many shared-memory consensus protocols has the **early binding** property identified by Aguilera and Toueg [2] as necessary to ensure correctness of a consensus protocol using a weak shared coin.

Chandra's protocol uses two arrays of bits to track the speed of processes with preference 0 or 1. The mechanism of the protocol is similar to previous protocols of Chor, Israeli, and Li [18] and Aspnes and Herlihy [9]: if a process observes that the other team has advanced beyond it, it adopts that value, and if it observes that all processes with different preferences are two or more rounds behind, it decides on its current preference secure in the knowledge that they will switch sides before they catch up. The arrays of bits effectively function as a max register, so it is natural to replace them with two max registers $m[0]$ and $m[1]$, initially set to 0, implemented as in Section 4. The complete description, pseudocode, and proof are given in the full version of the paper [3].

Theorem 2. *Let* SharedCoin_r*, for each* r*, be a shared coin protocol with constant agreement parameter, individual message complexity* $T_1(n)$*, and total message complexity* $T(n)$*. Then the algorithm described above implements a consensus protocol with expected individual message complexity* $O(T_1(n) + n)$ *and total message complexity* $O(T(n) + n^2)$*.*

Extension for General t. We can decrease the message complexity of the protocol by taking advantage of values of $t = o(n)$. The basic idea is to reduce message complexity by "deputizing" a set of $2t+1$ processes to run the protocol described above and produce an output value, which they broadcast to all other participants. For this, we fix processes $p_1, \ldots, p_{2t+1}$ to be the group of processes running the consensus protocol, which we call the *deputies*. When executing an instance of the protocol, each process first sends a $Start$ message to the deputies. If the process is a deputy, it waits to receive $Start$ notifications from $n - t$ processes. Upon receiving these notifications, the process runs

the consensus algorithm described above, where the only participants are processes $p_1, \ldots, p_{2t+1}$. Upon completing this protocol, each deputy broadcasts a $\langle Result, value \rangle$ message to all processes, and returns the decided value. If the process is not a deputy, then it simply waits for a $Result$ message from one of the deputies, and returns the corresponding value. Correctness follows from the previous arguments.

Theorem 3. *Let $n, t > 0$ be parameters such that $t < n$. The algorithm described above implements randomized consensus using $O(nt + t^2 \log^2 t)$ expected total messages, and $O(n + t \log^3 t)$ expected messages per process.*

8 Conclusions and Future Work

We have described a randomized algorithm for consensus with expected message complexity $O(n^2 \log^2 n)$ that tolerates $t < n/2$ crash faults; this algorithm also has the desirable property that each process sends and receives expected $O(n \log^3 n)$ messages on average, and message size is logarithmic. We also present a generalization that uses expected $O(nt + t^2 \log^2 t)$ messages.

Two conspicuous open problems remain. The first is whether we can close the remaining poly-logarithmic gap for the message cost of consensus in the classic model. Second, can we use techniques from this paper to help close the gap for message-cost of Byzantine agreement in the classic model? To the best of our knowledge, the current lower bound for message cost of Byzantine agreement is $\Omega(n^2)$, while the best upper bound is $O(n^{6.5})$ — a significant gap.

References

1. K. Abrahamson. On achieving consensus using a shared memory. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 291–302, New York, NY, USA, 1988. ACM.
2. M. K. Aguilera and S. Toueg. The correctness proof of Ben-Or's randomized consensus algorithm. *Distributed Computing*, 25(5):371–381, 2012.
3. D. Alistarh, J. Aspnes, V. King, and J. Saia. Communication-efficient randomized consensus. 2014. Full version available at http://www.cs.yale.edu/homes/aspnes/papers/disc2014-submission.pdf.
4. J. Aspnes. Lower bounds for distributed coin-flipping and randomized consensus. *J. ACM*, 45(3):415–450, May 1998.
5. J. Aspnes, H. Attiya, and K. 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, Aug. 2008.
6. J. Aspnes, H. Attiya, and K. Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM*, 59(1):2, 2012.
7. J. Aspnes and K. Censor. Approximate shared-memory counting despite a strong adversary. *ACM Transactions on Algorithms*, 6(2):1–23, 2010.
8. J. Aspnes and K. Censor-Hillel. Atomic snapshots in $O(\log^3 n)$ steps using randomized helping. In *Proceedings of the 27th International Symposium on Distributed Computing (DISC 2013)*, pages 254–268. 2013.
9. J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, Sept. 1990.

10. J. Aspnes and O. Waarts. Randomized consensus in expected $o(n \log^2 n)$ operations per processor. *SIAM J. Comput.*, 25(5):1024–1044, Oct. 1996.
11. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, Jan. 1995.
12. H. Attiya and K. Censor. Tight bounds for asynchronous randomized consensus. *J. ACM*, 55(5):20:1–20:26, Nov. 2008.
13. M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, pages 27–30, New York, NY, USA, 1983. ACM.
14. G. Bracha. An asynchronous [(n - 1)/3]-resilient consensus protocol. In *PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 154–162, New York, NY, USA, 1984. ACM.
15. G. Bracha and O. Rachman. Randomized consensus in expected O(n^2log n) operations. In S. Toueg, P. G. Spirakis, and L. M. Kirousis, editors, *WDAG*, volume 579 of *Lecture Notes in Computer Science*, pages 143–150. Springer, 1991.
16. T. D. 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.
17. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.
18. B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 86–97, New York, NY, USA, 1987. ACM.
19. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, Apr. 1988.
20. F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, Sept. 1998.
21. 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.
22. G. R. Grimmett and D. R. Stirzaker. *Probability and Random Processes*. Oxford University Press, 2001.
23. P. Hall and C. Heyde. *Martingale Limit Theory and Its Application*. Academic Press, 1980.
24. A. Karlin and A. Yao. Probabilistic lower bounds for byzantine agreement and clock synchronization. Unpublished manuscript.
25. R. Keralapura, G. Cormode, and J. Ramamirtham. Communication-efficient distributed monitoring of thresholded counts. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 289–300, New York, NY, USA, 2006. ACM.
26. V. King and J. Saia. Byzantine agreement in polynomial expected time. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 2013.
27. V. King and J. Saia. Faster agreement via a spectral method for detecting malicious behavior. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms(SODA)*, 2014.
28. L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.
29. M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, Apr. 1980.
30. M. Saks, N. Shavit, , and H. Woll. Optimal time randomized consensus - making resilient algorithms fast in practice. In *Proc. of the 2nd ACM Symposium on Discrete Algorithms (SODA)*, pages 351–362, 1991.