Communication-efficient randomized consensus

Dan Alistarh[1,2] · James Aspnes[3] · Valerie King[4,5,6] · Jared Saia[7]

Received: 20 April 2015 / Accepted: 10 September 2017 / Published online: 14 October 2017

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 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 [25,26] is arguably the most well-studied problem in distributed computing. The FLP impossibility result [23], showing that consensus could not be achieved determinist-

J. Aspnes: Supported in part by NSF Grants CCF-0916389, CCF-1637385, and CCF-1650596.

J. Saia: Supported in part by NSF CAREER Award 0644058 and NSF CCR-0313160.

A preliminary version of this work appeared in the Proceedings of the 28th International Symposium on Distributed Computing (DISC 2014).

Electronic supplementary material The online version of this article (doi:10.1007/s00446-017-0315-1) contains supplementary material, which is available to authorized users.

✉ Dan Alistarh
dan.alistarh@inf.ethz.ch

James Aspnes
aspnes@cs.yale.edu

[1] IST Austria, Klosterneuburg, Austria

[2] ETH Zurich, Zurich, Switzerland

[3] Department of Computer Science, Yale University, New Haven, CT, USA

[4] University of Victoria, Victoria, BC, Canada

[5] Simons Institute for the Theory of Computing, Berkeley, CA, USA

[6] Institute for Advanced Study, Princeton, NJ, USA

[7] Department of Computer Science, University of New Mexico, Albuquerque, NM, USA

cally 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. [21], employing failure detectors, e.g. [16], or by relaxing progress conditions to allow for *randomization*, e.g. [13].

In particular, a significant amount of research went into isolating time and space complexity bounds for randomized consensus in the shared-memory model, e.g. [3,5,9,10,12,14,22], 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 $\Theta(n)$ [7].[1] 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 in terms of message complexity is at least linear in n, the number of nodes. Specifically, to our knowledge, the best previously known upper bound for consensus in asynchronous message-passing requires expected $\Theta(n^3)$ messages, by simulating the elegant shared-memory protocol of Attiya and Censor-Hillel [12], using the simulation from [11]. 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.

Contribution 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.[2] 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 known 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 [15].

Further, we present a more efficient variant of the protocol for the case where t is known to be $o(n)$, 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. An interesting aspect of our constructions is that message sizes are small: since processes only communicate vote counts, whose meaning is clarified later, messages only require $O(\log n)$ bits of communication.

Technical background Since early work by Bracha and Rachman [14], implementations of weak shared coins for shared-memory systems with an adaptive adversary have generally been based on the idea of *voting*: each process generates locally votes of ± 1, communicates them, and then takes the sign of the sum of received votes. Generating a vote consists of flipping a local coin, and for the purposes of analysis, we track the sum of these generated votes as they are produced whether they are communicated immediately or not. In the simplest version of this voting method, if 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. Assuming a process writes out each vote before generating another, the adversary can only hide up to $f < n$ votes by delaying or crashing processes. So when the generated vote has absolute value at least n, 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 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

[1] We consider a model with n processes, $t < n/2$ of which may fail by crashing.

[2] In the following, logarithms are taken base 2.

is $\Theta(n^3)$ messages. Therefore, the question is whether this overhead can be eliminated.

Techniques To reduce both total and local message complexity, we employ two techniques. 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.

The vote-propagation mechanism is organized using a message-passing implementation of a shared object called a *max register* [6,8]. In brief, a max register is a shared object maintaining a value v, and supporting *MaxRead* and *MaxUpdate* operations, where *MaxRead* simply returns the value of the object, and $MaxUpdate(u)$ only updates u if $u > v$. Each cohort is assigned a max register implemented across the processes in that cohort. As long as fewer than half of those processes fail, the max register will report the most complete total of votes that has been written to it. Because processes will not generate new votes until they have propagated votes as required by the algorithm, we can show that the total in a cohort's max register is likely to be reasonably close to the actual total of all votes that have been generated by processes within the cohort. This is true even if the max register fails: non-faulty processes in the cohort will eventually stop generating new votes, and all but a few older votes that precede the failure will already have been propagated to some higher-level max register representing a larger cohort. In the analysis of the protocol, we show that even adding up all the sources of errors across all the cohorts still leaves the reported total in the root max register close to the actual sum of all of the votes that have ever been generated, whether those votes are announced or not.

Finally, we convert the shared coin construction into a consensus algorithm via a simple framework inspired by Chandra's shared-memory consensus protocol [15], which in turn uses ideas from earlier consensus protocols of Chor, Israeli, and Li [20] and Aspnes and Herlihy [9]. Roughly, we associate each of the two possible decision values for consensus with a message-passing implementation of a max register, whose value is incremented by the "team" of processes supporting that value for 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 value 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 gives the consensus protocol the same asymptotic complexity as the shared coin in expectation.

2 System model and problem statement

Asynchronous message-passing 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.

Adversary and cost 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.

Consensus 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.

A max register object 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.

3 Related work

The first shared-memory protocol for consensus was given by Chor et al. citeChorIL87 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. [3,10,14,27], 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].

A parallel line of research studied the message complexity of *synchronous* fault-tolerant consensus [17–19]. This work strongly relies on the fact that processors proceed in lock-step, and therefore the techniques would not be applicable in our setting.

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 the max register to be maintained by groups of $g < n$ processes.

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. 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.

To perform a MaxRead operation, the calling process communicates a *MaxRead*(R) request to all other processes, setting its local value v_i to be the maximum value received. Before returning this value, the process communicates an additional *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.*

The proof of this Lemma is virtually identical to the proof of correctness of the message-passing max register of [8, Theorem 5.1]. The only difference is that our construction may use a variable number of participants $g < n$; hence, we omit the proof here.

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.

```
1  Let $K = n^2 \log n$
2  Let $T = 4n \log n$
3  count ← 0
4  var ← 0
5  total ← 0
6  for $k$ ← 1, 2, ..., ∞ do
7      Let $w_k = 2^{\lfloor (k-1)/T \rfloor}$
8      Let vote = $\pm w_k$ 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 $n$ do
14         if $2^j$ does not divide $k$ then
15             break /* exit the loop */
16         Let $s$ be my level-$j$ ancestor, with children $s_\ell$ and $s_r$
17         in parallel do
               /* read left and right counts */
18             ⟨count$_\ell$, var$_\ell$, total$_\ell$⟩ ← ReadMax($R_{s_\ell}$)
19             ⟨count$_r$, var$_r$, total$_r$⟩ ← ReadMax($R_{s_r}$)
           /* update the parent */
20         WriteMax($s$, ⟨count$_\ell$ + count$_r$, var$_\ell$ + var$_r$, total$_\ell$ + total$_r$⟩)
21     if $n$ divides $k$ then
22         ⟨count$_{root}$, var$_{root}$, total$_{root}$⟩ ← ReadMax($R_{root}$)
           /* 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.

Vote propagation The key ingredient is a message-efficient construction of an approximate distributed vote counter, which allows processes to maintain an estimate of the total number of votes generated, and of their sum and variance. This 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 a tuple consisting of three values: the *count*, an estimate of the number of votes generated in the subtree; *var*, an estimate of the variance of the generated votes; and *total*, an estimate of the sum of generated votes. These tuples are ordered lexicographically, with the count as the most significant position. This ensures that processes always adopt a tuple with maximum *count*. As it happens, the correctness of the algorithm depends only on the fact that tuples with high counts overwrite tuples with lower counts, so the use of the other components to break ties is an arbitrary choice.

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 the level corresponding to the highest power of two which divides k (line 15). At each level from 1 (the level of the leaf's parent) up to r, the process reads the max registers of its 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).

Note that there is no guarantee that the reads of the children occur at the same time, or that the order in which different processes read two sibling registers are consistent: the only thing required for the analysis is that whatever value is ultimately stored in each max register has a high total count.

If n divides k, then the process also checks the count at the root. If the root variance 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 $T = 4n \log n$ 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.

After T loop iterations in an epoch, the weight of the votes doubles, and a new epoch begins. This ensures that only $O(\log n)$ epochs are needed until a single process can generate votes with 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 $K = n^2 \log n$, at which point the process returns the sign of the root total (line 24).

6 Complete algorithm analysis

In this section, we prove correctness of the weak shared coin algorithm in Sect. 5. Fix an adversary strategy, and given an execution of the algorithm under the control of this adversary, define $U^s[t]$ as the contents of the max register R_s corresponding to a subtree s at time t, for some particular linearization consistent with the observed execution ordering. We will compare this to an idealized value $V^s[t]$ that tracks all the votes that have been generated in s up to time t, whether or not they have been reported to anybody. The idea is that when a vote is generated by some process p, it immediately contributes its value to $V^s[t]$.total for all subtrees s that contain p, but may only contribute to $U^s[t]$.total at some later time, when this vote is finally propagated to R_s.

The variable $U^s[t]$ may depend in a complicated way on the schedule, because the adversary has a lot of freedom in choosing which values from smaller subtrees are combined together to produce values written to R_s. The variable $V^s[t]$ also depends on the schedule, because the weight of the next vote generated by some process in s depends on how many votes have previously been generated by that process, but it has an important property that $U^s[t]$ lacks: the sequence of values $V^s[t]$ form a *martingale*, which is a sequence of random variables $S_0, S_1, \ldots$ with the property that $\mathrm{E}\left[S_i \mid S_0, \ldots, S_{i-1}\right] = S_{i-1}$. The intuition is that even though the distribution in $V^s[t] - V^s[t-1]$ might depend on the choices of the adversary (which we take to be determined by the outcome of previous coin-flips in the algorithm), the expected increment produced by a fair w vote is zero.

What makes martingales useful for our purpose is that they generalize in many ways sums of independent random variables while allowing some limited dependence. Like sums of independent random variables, martingales have many well-understood convergence properties that constrain how far they are likely to drift from their starting points.

In particular, a martingale version of the Central Limit Theorem [24, Theorem 3.2] is used to show that, the generated vote $V^{\mathrm{root}}[t]$.total is normally distributed in the limit when $V^{\mathrm{root}}[t]$.var crosses the variance threshold K (Lemma 6). So with constant probability it will be far from the origin: the margin of victory of the majority sign will be large. As in previous shared-memory shared coin protocols, we next argue that this large margin means that when each process gets around to reading U^{root}.total$[t']$ at some later time t', it sees the same sign. This requires showing that the value of $V^{\mathrm{root}}[t']$.total doesn't change too much between t and t', and that the difference between $U^{\mathrm{root}}[t']$.total and $V^{\mathrm{root}}[t']$.total is also small.

Showing that $U^{\mathrm{root}}[t']$.total is close to $V^{\mathrm{root}}[t']$.total is the trickiest part of the analysis, and makes extensive use of a version of the Azuma–Hoeffding inequality from [10] (Lemma 8), which constrains how far a sequence of votes is likely to drift as a function of a bound on their total variance. To apply this inequality, we first observe that if s is a subtree with left and right subtrees ℓ and r, then $U^s[t]$ is always equal to $U^\ell[t_\ell] + U^r[t_r]$ for some earlier times t_ℓ and t_r. These times are when whatever process p wrote the value $U^s[t]$ to R_s previously read R_ℓ and R_r. The difference between $V^s[t]$ and $U^s[t]$ can then be accounted for by (a) the sum of all votes generated in ℓ and r during the intervals following t_ℓ and t_r, respectively; and (b) the sum of all votes generated in ℓ and r before t_ℓ and t_r that are not included in $U^\ell[t_\ell]$ and $U^r[t_r]$. Expanding out (b) recursively gives that the total error in $U^s[t]$ is equal to a sum of intervals of type (a) (Lemma 7). The rest of the argument involves using the length of these intervals to bound the variance of the votes included in them, and then apply Azuma–Hoeffding to show that these missing votes do not add too much error even if we allow the adversary to choose the boundaries of the intervals after seeing the votes.

We now give the details of the argument.

The first step in the analysis requires some straightforward lemmas about how many votes can be generated within a subtree before they are propagated to the max register at its root. For simplicity, we assume that the number of processes n is a power of two. If this is not the case, we build a sparse truncated tree, where each leaf still has height $\log n$, and maintain the property that the max register at a node is maintained by the nodes in the corresponding subtree.

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 *registers* 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 R_s at s in its loop iteration. See Fig. 1 for an illustration. We prove that this procedure has the following property:

Lemma 2 *Consider a subtree rooted at node s with m leaves, and let $x_1, x_2, \ldots, x_m$ be the number of votes most recently registered by member processes $q_1, q_2, \ldots, q_m$ at node s, respectively. Then the value of the* count *component of the max register R_s at s is at least $\sum_{j=1}^{m} x_j$.*

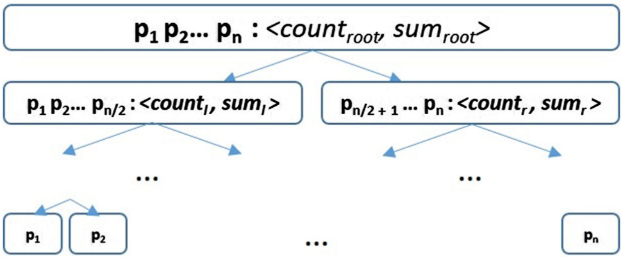
p₁ p₂… pₙ : <count_root, sum_root>
p₁ p₂… pₙ/₂ : <count_l, sum_l>
pₙ/₂ ₊ ₁ … pₙ : <count_r, sum_r>
…
…
p₁
p₂
…
pₙ

Fig. 1 Structure of the cohorts. A process maintains count and sum estimates for each cohort it is a member of

Proof We proceed by induction on the height of the subtree rooted at s, proving the following claim: Given the subtree rooted at s with corresponding processes $q_1, q_2, \ldots, q_m$, with $x_1, x_2, \ldots, x_m$ being the number of votes most recently registered by each process at node s, there exists a process q_i with $1 \leq i \leq m$ that performs a MaxUpdate with value at least $\sum_{j=1}^{m} x_j$ on the max register R_s when registering votes x_i at s.

If s is a leaf, the claim is straightforward, since each process writes the vote count, total, and variance at its leaf after generating the vote. (We can regard the single-writer register at the leaf as a max register.) For the induction step, let ℓ be the left child of s, and r be the right child. By the induction hypothesis, there exists a process p_i which updates the left child max register to $\sum_{j=1}^{m/2} x_j$, and a process p_k which updates the right child max register to $\sum_{j=m/2+1}^{m} x_j$.

We now focus on the value of the max register R_s at s after both processes p_i and p_k complete their MaxUpdate operations at s. Crucially, notice that, since both processes first read the max register R_ℓ at ℓ, then the one at r, one of the operations must necessarily "see" the value the other wrote at the corresponding child. Therefore, one of the processes, say p_i, must write a value of at least $\sum_{j=1}^{m} x_j$ to the max register R_s, as claimed. □

Technical results We now give a few technical results which will be useful in bounding the weight of votes generated by processors globally, and by individual processors. This will then be useful in bounding the individual message cost of the algorithm. Define n_i as the *number of votes* generated by process p_i during an execution. We first analyze the total variance of generated votes, the maximum weight of a vote, and the maximum number of votes generated by a processor.

Lemma 3 *Fix an arbitrary execution of Algorithm 1. Then the following hold:*

1. *The total variance of generated votes satisfies*

$$\sum_{i=1}^{n} \sum_{j=1}^{n_i} w_j^2 \leq \frac{K + 2n^2}{1 - 8n/T} = O(n^2 \log n). \tag{1}$$

2. *For any process p_i,*

$$w_{n_i} \leq \sqrt{1 + \frac{4K + 8n^2}{T - 8n}} = O(\sqrt{n}), \tag{2}$$

and

$$n_i = O(n \log^2 n). \tag{3}$$

Proof Examining the algorithm, we see that every n votes, process p_i both propagates all of its votes to the root max register and checks the total variance against K. It follows that (a) the root max register includes all but at most n votes for each process, and (b) each process generates at most n additional votes after $\mathsf{var}_{\mathrm{root}}$ passes K. Since each of these $2n$ missing and extra votes from process p_i has weight bounded by w_{n_i}, we get a bound

$$\begin{aligned}\sum_{i=1}^{n} \sum_{j=1}^{n_i} w_j^2 &\leq K + \sum_{i=1}^{n} 2n w_{n_i}^2 \\ &\overset{(a)}{\leq} K + \sum_{i=1}^{n} 2n \left(1 + \frac{4}{T} \sum_{j=1}^{n_i} w_j^2\right) \\ &\overset{(b)}{=} K + 2n^2 + \frac{8n}{T} \sum_{i=1}^{n} \sum_{j=1}^{n_i} w_j^2,\end{aligned}$$

where (a) follows by noticing that T consecutive weights are equal, and (b) follows by simple expansion. Rearranged, the previous relation gives (1). The asymptotic bound follows by calculation.

Next, we combine the preceding results to bound the maximum weight of any vote. We have

$$\begin{aligned}w_{n_i}^2 &\leq 1 + \frac{4}{T} \sum_{j=1}^{n_i} w_j^2 \\ &\leq 1 + \frac{4}{T} \cdot \frac{K + 2n^2}{1 - 8n/T} \\ &= 1 + \frac{4K + 8n^2}{T - 8n},\end{aligned}$$

where the second inequality follows from Eq. 1. The stated bound follows by taking the square root of both sides. Finally, we bound the total number of votes n_i generated by processor p_i during the execution. Notice that this is constrained by

$$w_{n_i} = 2^{\lfloor (n_i - 1)/T \rfloor} = O(\sqrt{n}),$$

where the latter bound follows by Eq. 2. This gives:

$$n_i = O(T \log(\sqrt{n})) = O(n \log^2 n),$$

which concludes the proof. □

We now turn our attention to the sum, over all processes, of the square of maximum votes. This will turn out to be useful later, and also gives a slightly better bound on the total number of generated votes than simply multiplying the bound in Eq. 3 by n.

Lemma 4 *For any execution of Algorithm 1, the following hold.*

1. *The squares of maximum votes satisfy*
$$\sum_{i=1}^{n} w_{n_i}^2 \le n + \frac{4K + 8n^2}{T - 8n} = O(n). \tag{4}$$
2. *The total number of generated votes satisfies*
$$\sum_{i=1}^{n} n_i = O(n^2 \log n). \tag{5}$$

Proof To prove the first claim, we notice that

$$\begin{aligned}\sum_{i=1}^{n} w_{n_i}^2 &\le \sum_{i} \left(1 + \frac{4}{T}\sum_{j=1}^{n_i} w_j^2\right)\\ &\le n + \frac{4}{T} \cdot \frac{K + 2n^2}{1 - 8n/T}\\ &= n + \frac{4K + 8n^2}{T - 8n}.\end{aligned}$$

For the proof of the second item, we notice that we have

$$\sum_{i=1}^{n} w_{n_i}^2 = \sum_{i} 4^{\lfloor (n_i - 1)/T \rfloor} \le an,$$

for sufficiently large n and some constant a, where the equality follows by definition, and the inequality follows by the previous claim. The floor is inconvenient, so we remove it by taking

$$\sum_{i=1}^{n} 4^{(n_i-1)/T-1} \le \sum_{i=1}^{n} 4^{\lfloor (n_i-1)/T \rfloor} \le an.$$

Convexity of $4^{(x-1)/T-1}$ implies that we can apply Jensen's inequality, to obtain

$$n \cdot 4^{\left(\frac{1}{n}\sum_{i=1}^{n} n_i - 1\right)/T-1} \le \sum_{i=1}^{n} 4^{(n_i-1)/T-1} \le an,$$

which gives

$$\sum_{i=1}^{n} n_i \le n(T(1 + \log_4 a) + 1) = O(n^2 \log n).$$

□

Tracking delayed votes With all this technical machinery in place, we now turn your attention to bounding the difference between the sum of votes generated in the tree, and the sum of votes visible at the root. More precisely, for any adversary strategy $\mathcal{A}$, let $\tau_{\mathcal{A}}$ be a stopping time corresponding to the first time t at which $U^{\text{root}}[t].\mathsf{var} \ge K$. We will use a martingale Central Limit Theorem to show that $V^{\text{root}}[\tau_{\mathcal{A}}].\mathsf{total}$ converges to a normal distribution as n grows, when suitably scaled. This will then be used show that all processes observe a pool of common votes whose total is likely to be far from zero.

In particular, we use the following simplified version of [24, Theorem 3.2], taken from [7]. 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.

Lemma 5 *Let $\{S_{mt}, \mathcal{F}_{mt}\}$, $1 \le t \le k_m$, $m \ge 1$ be a martingale array where for each fixed m, $\{S_{mt}, \mathcal{F}_{mt}\}$ is a zero-mean martingale with difference sequence $X_{mt} = S_{mt} - S_{m,t-1}$. If*

1. $\max_t |X_{mt}| \xrightarrow{p} 0$,
2. $\sum_t X_{mt}^2 \xrightarrow{p} 1$,
3. $\mathrm{E}\left[\max_t X_{mt}^2\right]$ *is bounded in m, and*
4. $\mathcal{F}_{m,t} \subseteq \mathcal{F}_{m+1,t}$ *for* $1 \le t \le k_m$, $m \ge 1$,

then $S_{mt} \xrightarrow{d} N(0, 1)$, where $N(0, 1)$ has a normal distribution with zero mean and unit variance.

Lemma 6 *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*

$$\frac{V^{\text{root}}[\tau_n].\mathsf{total}}{\sqrt{K}} \xrightarrow{d} N(0, 1). \tag{6}$$

Proof Let Y_{nt} be the t-th vote generated in the execution governed by $\mathcal{A}_n$, and let $X_{nt} = Y_{nt}/\sqrt{K}$ if $t \le \tau_n$ and 0 otherwise. Then

$$\frac{V^{\text{root}}[\tau_n].\mathsf{total}}{\sqrt{K}} = \sum_{t=1}^{\tau_n} Y_{nt}/\sqrt{K} = \sum_{t=1}^{\infty} X_{nt} = \lim_{t\to\infty} S_{nt}.$$

So to prove (6), we need only verify that the random variables X_{nt} satisfy the conditions of Lemma 5 for an appropriate choice of σ-algebras $\mathcal{F}_{n,t}$. That each X_{nt} has zero mean is immediate from their definition. Again following the approach of [7], we will let $\mathcal{F}_{n,t}$ be generated by the signs of the first t votes. We also let the i-th vote have the same sign for all values of n, so that $\mathcal{F}_{n,t} = \mathcal{F}_{n',t}$ for all n, n'. This trivially satisfies the last condition.

For the first condition, from Lemma 3 we have

$$\max_t |X_{nt}| \leq \max_i \frac{w_{n_i}}{\sqrt{K}} \leq \sqrt{\frac{1}{K} + \frac{4+8n^2/K}{T-8n}} = \sqrt{\frac{1}{n^2 \log n} + \frac{4+8/\log n}{4n\log n - n}}.$$

Since the denominator dominates the numerator in both fractions under the square root, this converges to zero always.

For the second condition, observe that $K \sum_t X_{nt}^2$ is the sum of the squares of the weights of all votes generated by τ_n. This quantity is at least K (as otherwise $U^{\text{root}}[\tau_n].\mathsf{var}$ could not have reached K), and is at most $\frac{K+2n^2}{1-8n/T}$ by Lemma 3. Dividing by K shows

$$1 \leq \sum_t X_{nt}^2 \leq \frac{1+2n^2/K}{1-8n/T} = O\left(\frac{1+O(1/\log n)}{1-O(1/\log n)}\right)$$

which converges to 1 always.

Finally, for the third condition, we have already shown that $\max_t |X_{nt}|$ converges to 0, and since $X_{nt}^2 = |X_{nt}|^2$ for all n, t, we have that X_{nt}^2 also converges to 0 for any fixed t; boundedness follows. □

Bounding unreported votes The next lemma characterizes the unreported votes that together make up the difference between $U^s[t_s].\mathsf{total}$ and $V^s[t_s].\mathsf{total}$ for each time t_s. Though t_s is an arbitrary time, we label it by the node s to easily distinguish it from similar earlier times corresponding to the contributions to $U^s[t_s]$ from the children of s. Observe that any value $U^s[t_s]$ for an internal node s is obtained from values $U^\ell[t_\ell]$ and $U^r[t_r]$ read from the left and right subtrees ℓ and r of s at some earlier times t_ℓ and t_r, these times being the linearization times of the read operations carried out by whatever process wrote $U^s[t_s]$. These subtree values are in turn constructed from the values of deeper subtrees (possibly by different processes), which means that for each proper subtree s' of s, there is some specific observation time $t_{s'}$ such that the value $U^{s'}[t_{s'}]$ is included in the computation of $U^s[t_s]$. We show that the votes omitted from $U^s[t_s]$ are precisely accounted for by summing the votes that were generated between time $t_{s'}$ and time $t_{\mathtt{parent}(s')}$ for each of these subtrees.

Lemma 7 *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 written to the max register corresponding to s at time t_s. For each subtree s' of s, define $t_{s'}$ as the time for which the value of $U^{s'}[t_{s'}]$ included in $U^s[t_s]$ was observed.*

Then

$$D^s[t_s] = \sum_{s'} \left(V^{s'}[t_{\mathtt{parent}(s')}].\mathsf{total} - V^{s'}[t_{s'}].\mathsf{total}\right), \quad (7)$$

where s' ranges over all proper subtrees of s.

Proof Let ℓ and r be the left and right subtrees of s, and let t_ℓ and t_r be the times at which the values added to produce $U^s[t_s].\mathsf{total}$ were read from these subtrees.

Then

$$\begin{aligned} D^s[t_s] &= V^s[t_s].\mathsf{total} - U^s[t_s].\mathsf{total} \\ &= (V^\ell[t_s].\mathsf{total} + V^r[t_s].\mathsf{total}) \\ &\quad -(U^\ell[t_\ell].\mathsf{total} + U^r[t_r].\mathsf{total}) \\ &= (V^\ell[t_s].\mathsf{total} + V^r[t_s].\mathsf{total}) \\ &\quad -(V^\ell[t_\ell].\mathsf{total} + V^r[t_r].\mathsf{total}) \\ &\quad +(D^\ell[t_\ell] + D^r[t_r]) \\ &= (V^\ell[t_s].\mathsf{total} - V^\ell[t_\ell].\mathsf{total}) \\ &\quad +(V^r[t_s].\mathsf{total} - V^r[t_r].\mathsf{total}) \\ &\quad +D^\ell[t_\ell] + D^{s_r}[t_r]. \end{aligned}$$

Iterating this recurrence gives

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

as claimed. □

To bound the right-hand side of (7), we consider each (horizontal) layer of the tree separately, and observe that the missing interval of votes $\left(V^{s'}[t_{\mathtt{parent}(s')}].\mathsf{total} - V^{s'}[t_{s'}].\mathsf{total}\right)$ 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 $2^h \left(1 + (4/T)\sum_{j=1}^{n_i} w_j^2\right)$, so if we sum the total variance of at most 2^h votes from all processes, we get at most

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

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 $D^{\text{root}}[t]$.

We will use the following extended version of the classic Azuma–Hoeffding inequality from [10]:

Lemma 8 ([10, Theorem 4.5]) *Let $\{S_0, \mathcal{F}_0\}$, $0 \le i \le n$ 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, $|X_i| \le w_i$ with probability 1; and that there exists a bound W such that $\sum_{i=1}^{n} w_i^2 \le W$ with probability* 1. *Then for any $\lambda > 0$,*

$$\Pr\left[S_n \ge \lambda\right] \le e^{-\lambda^2/2W}. \tag{8}$$

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) $\left(\sum_{j<i}(Y_j^{tsW})^2\right) + (X_i^s)^2 \le W$; 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 $w_i = (Y_i^{tsW})^2$ for the purposes of Lemma 8, and from (b) we have $\sum w_i^2 \le W$ always. It follows that, for any $c > 0$,

$$\Pr\left[\sum_{i=1}^{n} Y_i^{tsW} \ge \sqrt{2cW \ln n}\right] \le e^{-c\ln n} = n^{-c}.$$

There are polynomially many choices for t, s, and W; taking a union bound over all such choices shows that, for c sufficiently large, with high probability $\sum_{i=1}^{n} Y_i^{tsW}$ is bounded by $\sqrt{2cW \ln n}$ for all such intervals.

We can use this to show:

Lemma 9 *For any adversary strategy and sufficiently large n, with probability $1 - o(1)$, it holds that at all times t,*

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

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

Suppose that for each t, s, W, it holds that $Y^{tsW} \le \sqrt{9W \ln n}$. 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

$$\sum_s (V^s[t_{\mathtt{parent}(s)}] - V^s[t_s]) = \sum_{h=0}^{\log n-1} \sum_{s,|s|=2^h} (V^s[t_{\mathtt{parent}(s)}] - V^s[t_s])$$
$$= \sum_{h=0}^{\log n-1} \sum_{s,|s|=2^h} Y^{t_s s W_s}.$$

Define W_s as the total variance of the votes generated by s in the interval $(t_s, t_{\mathtt{parent}(s)}]$. Then we have

$$\sum_{h=0}^{\log n-1} \sum_{s,|s|=2^h} Y^{t_s s W_s} \le \sum_{h=0}^{\log n-1} \sum_{s,|s|=2^h} \sqrt{2cW_s \ln n}$$
$$= \sqrt{2c \ln n} \sum_{h=0}^{\log n-1} \sum_{s,|s|=2^h} \sqrt{W_s}.$$

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 between $U^{\mathsf{root}}[t].\mathsf{total}$ and $V^{\mathsf{root}}[t].\mathsf{total}$, 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 p_i corresponding to the subtree rooted at s generates at most 2^h votes, and each such vote has variance at most $w_{n_i}^2$, we have

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

Furthermore, the subtrees at any fixed level h partition the set of processes, so applying Lemma 4 gives

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

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

$$W_s \le \tfrac{2^h}{n} \cdot 2^h \left(n + \tfrac{4K+8n^2}{T-8n}\right)$$
$$= 2^{2h} \left(1 + \tfrac{4K+8n^2}{Tn-8n^2}\right)$$

and thus

$$\sqrt{W_s} \le 2^h \sqrt{1 + \frac{4K + 8n^2}{Tn - 8n^2}}$$

which gives the bound

$$\begin{aligned}
&\sqrt{9\ln n}\sum_{h=0}^{\log n-1}\sum_{s,|s|=2^h}\sqrt{W_s}\\
&\quad\le\sqrt{9\ln n}\sum_{h=0}^{\log n-1}2^h\sqrt{1+\frac{4K+8n^2}{Tn-8n^2}}\\
&\quad=\sqrt{9\ln n\left(1+\frac{4K+8n^2}{Tn-8n^2}\right)}\sum_{h=0}^{\log n-1}2^h\\
&\quad=3(n-1)\sqrt{\ln n}\sqrt{1+\frac{4K+8n^2}{Tn-8n^2}}.\\
&\quad=3(n-1)\sqrt{\ln n}\sqrt{1+\frac{\log n+2}{\log n-2}}\\
&\quad\le 6n\sqrt{\log n},
\end{aligned}$$

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 $3\sqrt{2/\ln 2}<6$. □

Bounding extra votes For the last step, 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

$$\Pr\left[\exists i\le n:|S_i|\ge\lambda\right]\le\frac{\lambda^2}{\mathrm{Var}[S_n]}.$$

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,

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

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

Putting the pieces together Finally, from Lemma 6, we have that the total common vote $V^{\mathsf{root}}[\tau_n].\mathsf{total}$ converges in distribution to $N(0,1)$ when scaled by $\sqrt{K}=n\sqrt{\log n}$. In particular, for any fixed constant c, there is a constant probability $\pi_c>0$ that for sufficiently large n, $\Pr\left[V^{\mathsf{root}}[\tau_n]\ge cn\sqrt{\log n}\right]\ge\pi_c$.

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

If this occurs, then every process observes, for some t, $U^{\mathsf{root}}[t].\mathsf{total}\ge 7n\sqrt{\log n}-6n\sqrt{\log n}-o(n\sqrt{\log n})>0$. 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 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 done by q. Each such process q performs at most $O(n\log^2 n)$ votes, and there are 2^h such 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 bound of $O(n\log^3 n)$ messages per process. Applying the same reasoning to the bound on the total number of votes from Lemma 4 gives the bound of $O(n^2\log^2 n)$ on total message complexity. □

7 From a shared coin to consensus

For the last step, we must show how to convert a message-efficient weak shared coin into message-efficient consensus. Typically, a shared coin is used in the context of a larger framework in which faster processes do not use the coin (to ensure validity) while slower processes use the shared coin (to ensure agreement after each round with constant probability). A direct approach is to drop the shared coin into an existing message-passing agreement protocol like Ben-Or's [13]. However, as recently observed by Aguilera and Toueg [2], this can produce bad outcomes when the failure bound is large, because the adversary may be able to predict the value of the shared coin before committing to the value of the faster processes. Instead, we adapt a shared-memory consensus protocol, due to Chandra [15], which, like many shared-memory consensus protocols has the *early binding* property identified by Aguilera and Toueg as necessary to avoid this problem.

Chandra's protocol uses two long 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 [20] 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 Sect. 4. Pseudocode for the resulting algorithm is given in Algorithm 2.[3]

```
$x \leftarrow$ input
for $r \leftarrow 1 \ldots \infty$ do
    WriteMax($m[x], r$)
    $r' \leftarrow$ ReadMax($m[1-x]$)
    if $r' \geq r+1$ then
        $x' \leftarrow 1-x$
    else if $x' = x$ then
        $x' \leftarrow$ SharedCoin$_r$()
    else if $x' = x-1$ then
        $x' \leftarrow x$
    else
        return $x$
    $x'' \leftarrow$ ReadMax($m[x]$)
    if $r'' < r+1$ then
        $x \leftarrow x'$
```

Algorithm 2: Consensus protocol

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 Algorithm 2 implements a consensus protocol with expected individual message complexity* $O(T_1(n)+n)$ *and total message complexity* $O(T(n)+n^2)$*.*

Proof Validity follows from the fact that, if all processes start with the same preference x, then $m[1-x]$ never advances beyond zero and no process ever changes its preference.

Agreement is more subtle, although the ability to read the max register atomically makes the proof easier than the original proof in [15]. Suppose that some process returns p. Then this process previously read $m[1-x] \leq r-2$ after writing $m[x] \geq r$. At the time of this read, no other process has yet written $r-1$ or greater to $m[1-x]$, so any other process with preference $1-x$ has not yet executed the write at the start of round $r-1$. It follows that each such process sees $m[x] \geq (r-1)+1$ and adopts x as its new preference before reaching round r. Since all processes entering round r agree on x, they all decide x after at most one more round.

For termination, it cannot be the case that processes with different preferences both avoid the shared coin in round r, because processes with both preferences x would need to read $m[1-x] < r$ after writing $m[x] = r$, violating the specification of the max register. Suppose that at least one process with preference x does not execute the shared coin in round r, and processes with preference $1-x$ do. For this to occur, the adversary must show the preference-x process $m[1-x] \leq r-1$ before any preference-$(1-x)$ process writes r to $m[1-x]$, and thus before any process (with either preference) initiates SharedCoin$_r$. It follows that SharedCoin$_r$ returns x to all processes with constant probability, producing agreement in round r and termination after at most two more rounds.

For a shared coin protocol with agreement parameter δ, we have a probability of at least δ per round that the protocol reaches agreement in that round. This gives an expected number of rounds of at most $1/\delta + 2 = O(1)$. Each involves at most three max-register operations for each process, imposing a cost of $O(n)$ messages on that process and $O(1)$ messages on each other process, plus an invocation of the shared coin. The expected individual message complexity is thus $O(n) + T_1(n) = O(T_1(n)+n)$ as claimed. A similar argument shows the bound for total message complexity. □

Using Algorithm 1 for the shared coin, $T(n)$ dominates, giving that each process sends and receives $O(n \log^3 n)$ messages on average.

8 Variant for general failure parameter *t*

In this section, we describe a variant of the consensus protocol which adapts to the number of failures t. The basic idea is to reduce message complexity by "deputizing" a set of $2t+1$ processes to run the consensus protocol described in Sect. 7 and produce an output value, which they broadcast to all other participants. We note that the failure upper bound t needs to be known in advance by the protocol.

Description 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 algorithm from Sect. 7, where the only par-

[3] The presentation here is influenced by the simplified version of Chandra's original protocol given in [4].

ticipants are processes $p_1, \ldots, p_{2t+1}$. Upon completing this protocol, each deputy broadcasts a $\langle \mathit{Result}, \mathit{value} \rangle$ 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.

Analysis This construction guarantees the following properties. The proof follows from the argument in Sect. 7.

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

9 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.

Acknowledgements Open access funding provided by Institute of Science and Technology (IST Austria).

References

1. Abrahamson, K.: On achieving consensus using a shared memory. In: Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC '88, pp. 291–302, New York, NY, USA. ACM (1988)
2. Aguilera, M.K., Toueg, S.: The correctness proof of Ben-Or's randomized consensus algorithm. Distrib. Comput. **25**(5), 371–381 (2012)
3. Aspnes, J.: Lower bounds for distributed coin-flipping and randomized consensus. J. ACM **45**(3), 415–450 (1998)
4. Aspnes, J.: Randomized protocols for asynchronous consensus. Distrib. Comput. **16**(2–3), 165–175 (2003)
5. Aspnes, J., Attiya, H., Censor, K.: 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, pp. 325–334 (2008)
6. Aspnes, J., Attiya, H., Censor-Hillel, K.: Polylogarithmic concurrent data structures from monotone circuits. J. ACM **59**(1), 2 (2012)
7. Aspnes, J., Censor, K.: Approximate shared-memory counting despite a strong adversary. ACM Trans. Algorithms **6**(2), 1–23 (2010)
8. Aspnes, J., Censor-Hillel, K.: Atomic snapshots in $O(\log^3 n)$ steps using randomized helping. In: Proceedings of the 27th International Symposium on Distributed Computing (DISC 2013), pp. 254–268 (2013)
9. Aspnes, J., Herlihy, M.: Fast randomized consensus using shared memory. J. Algorithms **11**(3), 441–461 (1990)
10. Aspnes, J., Waarts, O.: Randomized consensus in expected $O(n \log^2 n)$ operations per processor. SIAM J. Comput. **25**(5), 1024–1044 (1996)
11. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM **42**(1), 124–142 (1995)
12. Attiya, H., Censor, K.: Tight bounds for asynchronous randomized consensus. J. ACM **55**(5), 20:1–20:26 (2008)
13. Ben-Or, M.: 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, pp. 27–30, New York, NY, USA. ACM (1983)
14. Bracha, G., Rachman, O.: Randomized consensus in expected O(n^2log n) operations. In: Toueg, S., Spirakis, P.G., Kirousis, L.M. (eds.) WDAG, volume 579 of Lecture Notes in Computer Science, pp. 143–150. Springer (1991)
15. Chandra, T.D.: Polylog randomized wait-free consensus. In: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 166–175, Philadelphia, Pennsylvania, USA, 23–26 May 1996
16. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (1996)
17. Chlebus, B.S., Kowalski, D.R.: Robust gossiping with an application to consensus. J. Comput. Syst. Sci. **72**(8), 1262–1281 (2006)
18. Chlebus, B.S., Kowalski, D.R.: Time and communication efficient consensus for crash failures. In: Dolev, S. (ed.) Distributed Computing, pp. 314–328. Springer, Berlin (2006)
19. Chlebus, B.S., Kowalski, D.R., Strojnowski, M.: Fast scalable deterministic consensus for crash failures. In: Proceedings of the 28th ACM symposium on Principles of distributed computing, pp. 111–120. ACM (2009)
20. Chor, B., Israeli, A., Li, M.: On processor coordination using asynchronous hardware. In: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC '87, pp. 86–97, New York, NY, USA. ACM (1987)
21. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (1988)
22. Fich, F., Herlihy, M., Shavit, N.: On the space complexity of randomized synchronization. J. ACM **45**(5), 843–862 (1998)
23. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985)
24. Hall, P., Heyde, C.C.: Martingale Limit Theory and Its Application. Academic Press, Cambridge (1980)
25. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. **4**(3), 382–401 (1982)
26. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. J. ACM **27**(2), 228–234 (1980)
27. Saks, M., Shavit, N., Woll, H.: Optimal time randomized consensus—making resilient algorithms fast in practice. In: Proceedings of the 2nd ACM Symposium on Discrete Algorithms (SODA), pp. 351–362 (1991)