Achieving Sublinear Complexity under Constant T in T-interval Dynamic Networks

Ruomu Hou[*]
National University of Singapore
Republic of Singapore
houruomu@comp.nus.edu.sg

Irvan Jahja[*]
National University of Singapore
Republic of Singapore
irvan@comp.nus.edu.sg

Yucheng Sun[*]
National University of Singapore
Republic of Singapore
suny@u.nus.edu

Jiyan Wu[*]
National University of Singapore
Republic of Singapore
wujiyan@nus.edu.sg

Haifeng Yu[*]
National University of Singapore
Republic of Singapore
haifeng@comp.nus.edu.sg

ABSTRACT

This paper considers standard T*-interval dynamic networks*, where the N nodes in the network proceed in lock-step *rounds*, and where the topology of the network can change arbitrarily from round to round, as determined by an *adversary*. The adversary promises that in every T consecutive rounds, the T (potentially different) topologies in those T rounds contain a common connected subgraph that spans all nodes. Within such a context, we propose novel algorithms for solving some fundamental distributed computing problems such as Count/Consensus/Max. Our algorithms are the first algorithms whose complexities do not contain an $\Omega(N)$ term, under constant T values. Previous sublinear algorithms require significantly larger T values.

1 INTRODUCTION

Model and assumptions that we adopt. This paper considers standard *synchronous* T*-interval dynamic networks* [2–4, 9, 12, 16, 22]. Under this *synchronous* timing model, nodes in the network proceed in lock-step *rounds*, all starting from round 1. The topology of such a dynamic network can change arbitrarily from round to round, as determined by an *adversary*. The algorithm does not know the topology in each round. Let X_i denote the topology in round i ($i \geq 1$). We consider *oblivious adversaries* as in [3, 4, 12, 15], where the adversary determines all the X_i's for $i \geq 1$, before the algorithm starts execution and without the adversary seeing the outcomes of the random coin flips in the algorithm. Under the T-interval model, the adversary promises that for every $i \geq 1$, the topologies X_i through X_{i+T-1} contain a common connected subgraph that spans all nodes. Hence one can intuitively view $\cap_{j=i}^{i+T-1} X_j$ as a stable "backbone" of the network in those T rounds. (The algorithm does not know $\cap_{j=i}^{i+T-1} X_j$.) Except this constraint, the topologies X_i through X_{i+T-1} can still be arbitrarily different from each other. Smaller T implies the network being more dynamic, and makes it harder to design algorithms. Despite the changing topologies, the set of nodes in the system is always fixed, and there are total N nodes. The algorithm knows neither N nor any upper bound on N. Each node has a unique *id* of size $O(\log N)$.[1] Without loss of generality, we assume all ids are non-zero. For sending/receiving messages, we consider the local broadcast CONGEST model as in [1, 2, 16, 27]: In each round, a node first does some local processing, and then chooses a single message of size $O(\log N)$ and sends that single message to all its neighbors. (A node cannot send different messages to different neighbors.) At the end of each round, a node receives all the messages sent by all its neighbors in that round. A node does not know its neighbors in a round, until after it receives messages from those neighbors.

For a given dynamic network, we say that $u \xrightarrow{r} w$, if either $w = u$ or node w is a neighbor of node u in round r. The *dynamic diameter* [12, 22] (denoted as D) of a dynamic network is defined to be the smallest D, such that for all $r \geq 1$ and all nodes u and w, there exists x_1 through x_{D-1} where $u \xrightarrow{r} x_1 \xrightarrow{r+1} x_2 \xrightarrow{r+2} \dots \xrightarrow{r+D-1} w$. Intuitively, this means that every node can causally influence every other node within D rounds, which is a natural generalization of diameter in static networks. For a T-interval dynamic network, D can range from 1 to $N-1$. Note that the algorithm knows neither D nor any upper bound on D.

Problems we consider. Within such context, we consider a number of fundamental distributed computing problems:

- Count: All nodes aim to output N.
- LeaderElection: All nodes should output the id of some common node (i.e., the leader).
- Consensus: Each node starts with some input of $O(\log N)$ size. All nodes should output some common decision value, which must be one of the N inputs.
- Max/Sum: Each node starts with some input of $O(\log N)$ size. All nodes should output the maximum/sum of all the N input values.
- ConfirmedFlood [31]: One designated node wants to disseminate its $O(\log N)$-size input to all N nodes. It should output "done" after all nodes have received its input. Other nodes output nothing.

We will use *time complexity* as the measure of goodness of an algorithm, which is the number of *rounds* needed for all nodes to output (or the designated node to output, for the ConfirmedFlood problem) and further terminate in that algorithm, under the *worst-case* input and *worst-case* adversary.

[*]The authors are alphabetically ordered.
[1]Having such ids do not give nodes a polynomial upper bound on N, because only the largest id among the N nodes translates to a polynomial upper bound on N. Determining the largest id in the system is non-trivial, and is at least as hard as the LeaderElection problem (see later) that we intend to solve.

Existing results on these problems. Except [17], existing approaches [12, 16, 22] for solving the above problems under our settings all have $\Omega(N)$ time complexity,[2] regardless of the value of T. Most of these prior works actually focus on the *token dissemination* problem [12, 16], and then the earlier problems can be solved as a by-product of solving token dissemination. In token dissemination, each node starts with some token/input of $O(\log N)$ size, and the goal is to let every node see all N tokens/inputs. The nodes can then apply various functions on the N inputs to solve all the earlier problems. Obviously, with maximum message size of $\Theta(\log N)$, a constant-degree node always needs $\Omega(N)$ rounds to get all the tokens.

When the dynamic diameter D is large (e.g., $D = \Theta(N)$), it is obvious that solving the above problems requires $\Omega(N)$ rounds, regardless of whether we take the token dissemination route. The reason is simply that all these problems fundamentally entail each node "hearing from" at least a majority of all the N nodes, hence requiring $\Omega(D)$ rounds. On the other hand, when the dynamic diameter D is small (e.g., $D = O(\log N)$), there is no lower bound preventing us from achieving $o(N)$ complexity.[3]

In fact, recently, Jahja and Yu [17] have proposed sublinear-complexity algorithms for solving these problems, with a time complexity of $O(\bar{D}^3 \log^2 N)$. Here $\bar{D}$ is the *backbone diameter*[4] of the T-interval dynamic network. These are the only algorithms known so far, whose time complexities do not contain an $\Omega(N)$ term. Unfortunately, these algorithms only work when $T \geq c_0\bar{D}^2 \log^2 N$ for some positive constant c_0. In fact, their algorithmic framework [17] fundamentally entails $T = \Omega(\bar{D})$: Their framework first computes certain paths of length up to $\bar{D}$ in the network, and then transfers the value from each node along such pre-computed paths, with proper aggregation along the way. The value of T needs to be $\Omega(\bar{D})$, so that some of these paths can be stable for $\Omega(\bar{D})$ rounds and do not get disrupted, after the paths are computed and before the transfer completes.

Our main results. This paper explores, *under constant T values*, the possibility of solving the earlier problems in sublinear complexity with respect to N. (Smaller T makes it harder to design algorithms.) To this end, we propose novel randomized algorithms for solving Count/LeaderElection/Consensus/Max/Sum/ConfirmedFlood with $O(DN^{\frac{6}{7}}\text{polylog}(N))$ time complexity, as long as $T \geq 4$ and the required error probability $\epsilon = \Omega(\frac{1}{\text{poly}(N)})$ for some polynomial $\text{poly}(N)$.

Compared to the sublinear-complexity algorithm in [17] that requires $T \geq c_0\bar{D}^2 \log^2 N$, we believe that our results for such constant T values shed new light on the complexities of these problems in dynamic networks, since smaller T values bring out the full dynamism in dynamic networks. Designing sublinear algorithms or obtaining linear lower bounds for $1 \leq T \leq 3$ will be our future work.

General approach (for which we do not claim novelty or contribution). Let us use our Count algorithm as an example to explain i) the general approach we take, and ii) our novel techniques. We take the following general approach: Our Count algorithm first lets each node become a *sink* with about $\frac{\log N}{N^{1/7}}$ probability independently,[5] which will result in a total of about $N^{\frac{6}{7}} \log N$ sinks. Each of N nodes next starts with a value of 1.0, and does a random walk from itself. (Our final results will not depend on mixing time.) A node's value gets transferred hop by hop along the walk. Two walks merge if they collide, and the values carried by them also get summed up. Once a value reaches a sink, it stays there. Note that during this process, the sum of all the values in the system is always N. Finally, every sink sums up all the values it receives, and disseminates the sum to all nodes. Every node then adds up all the sums that it has seen from all the sinks, as the final count. The final count will be exactly N if: i) the value from every node reaches some sink, and ii) all sinks successfully disseminate their respective sums to all nodes. Compared to the algorithmic framework in [17], this approach does not have stringent requirements on T.

Our novel techniques. The above general approach has been widely used before (e.g., [3]) in dynamic networks, and we do not claim any novelty or contribution for that. Prior works however do not solve Count (or other problems we consider) in sublinear complexity, regardless of how small D is. In fact, a naive implementation of this approach would require prior knowledge of N and D, and would furthermore result in quite large time complexity. Our contributions hence are the following novel techniques, which enable computing Count in sublinear-complexity via this well-known approach:

- (Elaborated in Section 3.1 and 3.2.) We need each random walk to reach some sink fast. Researchers have mainly considered two types of random walks in dynamic networks: *simple random walks* [5–8, 13, 29] and *max-degree random walks (lazy random walk)* [3, 7, 8, 14]. Unfortunately, both types will take too long to reach sinks. We propose a simple generalization, called *ρ-bounded-probability random walk* (or *ρ-BRW*), which can reach sinks faster with proper ρ.
  Next, implementing random walks in a T-interval dynamic network is generally difficult, because in each round a node u knows its neighbors only *after* it has received messages from them. Hence u cannot simply examine its neighbors, and then direct the walk to some random neighbor: By the time this happens, u's neighbors may have already changed. Existing works have not dealt with this issue: They either analyze the random walk without worrying about implementation [7, 8, 14, 29], or they assume each node knows its neighbors beforehand [3, 5, 6, 13] (which would deviate from the standard T-interval model). In our design, when node u tries to direct the walk to node v, if v is no longer

[2] Some of these algorithms are designed for *adaptive adversaries* instead of for oblivious adversaries. Adaptive adversaries are harder to deal with than oblivious ones. The complexity of these algorithms remains $\Omega(N)$ under oblivious adversaries.

[3] We note that existing algorithms [12, 16, 22] for these problems typically describe their complexities as a function of N, instead of as a function of both N and D. Nevertheless, their complexities remain $\Omega(N)$ even if D is as small as $O(1)$. Hence even if one were to describe their complexities as a function of both N and D, that function would still contain a $\Omega(N)$ term.

[4] Specifically, let H_r be the stable backbone of the dynamic network that persists from round r through $r + T - 1$. Note H_r is a static graph. Then $\bar{D}$ is the maximum diameter among all these H_r's, for $r \geq 1$. When $T \geq \bar{D}$, the dynamic diameter D is always no larger than the backbone diameter $\bar{D}$.

[5] Naively doing these steps would require knowledge of N and D. We overcome this key difficulty later.

u's neighbor, we let the walk remain on u. We show that the resulting behavior still maps to a proper ρ-BRW on some (new) dynamic network that is derived from the original dynamic network. As a caveat, this no longer works when u's degree is too large — we deal with that separately.

- (Elaborated in Section 3.3 and 3.4.) We propose a novel and elegant *soundness checking mechanism* to enable a node w to determine, in sublinear complexity, whether it has "heard from" all nodes in the system. Since such functionality is rather basic/fundamental, our mechanism can be potentially useful elsewhere too. Our mechanism first lets the nodes distribute/replicate *shares* (as well as *inverse* of these shares) among themselves, where each share is chosen randomly from the group $\mathbb{Z}_p$ for some prime number p. This is done in such a way that i) all shares in the system add up to 0, and ii) if no node has a degree larger than or equal to p, then for any given subset of the nodes, with probability $1 - \frac{1}{p}$, the sum of the shares held by those nodes will not be 0. Our Count algorithm then computes the sum of all these shares, together with the sum of all the initial 1.0 values from the nodes. This means that node w "sees" a certain value iff w "sees" the corresponding shares. Finally, w can judge whether it has "heard from" all nodes, based on whether all the shares it sees add up to 0.
  We are not aware of any existing work that uses the idea of random shares for solving Count or related distributed computing problems. Random shares have been widely used for secret sharing (e.g., [18, 30]), where for example, all shares are needed to recover the secret, while any subset of the shares does not leak the secret. Secret sharing aims for security/privacy, and has been mostly studied on static networks with clique topologies. Because of this, our way of distributing/replicating the shares in dynamic networks with arbitrary topologies, as well as our final guarantees, is quite different from secret sharing.
- (Elaborated in Section 3.5.) In our algorithm, some node u (e.g., if u is close to all the sinks) may pass the soundness checking much earlier than some other node w, in which case u may output the final count much earlier than w. This makes it challenging for the nodes to properly terminate: If node u immediately terminates upon outputting the final count, then node w may not be able to compute the correct count any more. In fact, the early termination of all such u's could even partition the network. It is far from obvious how to overcome this: It seems that node u should terminate only after all other nodes have already output. To do so, the system needs to count the number of nodes that have already output. But during this second counting process, again, node u may complete the counting faster and then terminate, before node w completes the counting. This again causes problem for w.
  We propose a novel design of *dual-schedule termination*, to enable the nodes to properly and efficiently terminate. When a node u first outputs the final count in round t, it schedules a *slow termination* for all nodes in the system, where the slow termination will happen in round $t + N$. Next node u invokes a trivial variant of our Count algorithm to compute the total number M of nodes that have already received such scheduling. When nodes u sees $M = N$, it will schedule a second *fast termination* for all nodes in the system. With this design, it is still possible for a node to terminate, when some other nodes are still running the algorithm. Nevertheless, we will show that i) the termination of one node will never affect the correctness of the algorithm running on another node, and ii) all nodes must terminate within a constant factor of the time needed for all nodes to output.

2 ADDITIONAL RELATED WORKS

This section discusses some additional related works, beyond those covered in Section 1.

Ahmadi et al. [3] propose a random-walk-based algorithm for token dissemination in dynamic networks. They consider the *unicast* model, where each node knows its neighbors beforehand and can send different messages to different neighbors. In comparison, we assume that neighbors are not known beforehand and that each node can only send the same message to all neighbors. They consider adaptive adversaries and all $T \geq 1$, while we assume oblivious adversaries and $T \geq 4$. They focus on message complexity instead of time complexity, and their algorithm has $\tilde{\Omega}(N^2)$ time complexity. Since their model is quite different from ours, their results are not directly comparable to ours. Nevertheless, our algorithms and theirs share the same general approach: They also select multiple sinks, collect the tokens to the sinks by doing random walks, and finally the sinks disseminate whatever they have collected. As explained earlier, we do not claim contribution in this general approach. Instead, we claim novelty in our techniques that enable computing Count in sublinear-complexity via this general approach: First, our novel soundness checking mechanism enables our algorithm, among other things, to overcome the challenge of doing random walks and determining the appropriate number of sinks when N is *unknown*. In comparison, Ahmadi et al.'s algorithm assumes the prior knowledge of N. Second, we use ρ-BRW with $\rho = 1/N^{\frac{1}{7}}$, while Ahmadi et al.'s algorithm uses lazy random walk. Finally, on large-degree nodes, we try to direct the walk to a sink neighbor, to overcome the issue of unknown neighbors for those nodes. Their algorithm does this as well, but for a different purpose of reducing message complexity.

Augustine et al. [5, 6] propose interesting algorithms for byzantine agreement and leader election on dynamic networks. Their algorithms use simple random walks to sample nodes in the network, and assume the topologies to be fast-mixing expander graphs. In comparison, our algorithm does not need any assumption on mixing time. On the other hand, their algorithms allow node churn, while our algorithm assumes a static set of nodes.

There have also been works on counting and consensus in dynamic networks whose topology may be disconnected. For example, Brandes et al. [9] propose algorithms for counting (via token dissemination) in $\tilde{O}(N^2)$ rounds, in T-interval dynamic networks where the edges may fail independently and potentially result in disconnected topologies. Chlebus et al. [11] propose several interesting deterministic algorithms for solving consensus in dynamic networks that can be disconnected. With unknown N and D, these algorithms can achieve sublinear time complexity only when the network allows large message size such as $\Theta(N \log N)$. In comparison, our algorithm uses only $\Theta(\log N)$ message size, but our algorithm does not work when the topologies can be disconnected or when $T < 4$. Hence their end results and ours are not directly comparable. Also because of the rather different settings, the techniques from [9, 11] cannot be directly used to achieve our goal in our setting.

The prior works discussed so far, as well as our work, all assume that the nodes in the dynamic network have unique ids. In a completely separate line of research [10, 19–21, 23–26], researchers have obtained many elegant results on counting the number of nodes in *anonymous* dynamic networks where nodes do not have unique ids. Not having unique ids makes the problem significantly harder, and it is not surprising that these algorithms all require $\Omega(N)$ complexity. In fact, the lack of unique ids is a central challenge in solving Count in anonymous networks, which sometimes even renders the Count problem unsolvable. Our Count algorithm fundamentally relies on the unique ids of the nodes, and does not work in anonymous networks. Hence our results and these prior results in anonymous networks are not comparable — they are simply for two fundamentally different settings. In terms of algorithmic techniques, similar to our algorithms, many of these existing algorithms let each node start with a pre-determined value (e.g. 1.0). These values will gradually "diffuse" and be "absorbed" by some *sinks/leaders*, and finally the sinks/leaders may potentially broadcast the absorbed values. As explained earlier, we do not claim any contribution in this well-known general approach. Instead, we claim novelty in our specific way of doing our ρ-BRW, in our novel soundness checking mechanism, and in our dual-schedule termination. None of these key techniques has been used in those prior works in anonymous dynamic networks, perhaps partly because these techniques rely on unique node ids.

3 KEY IDEAS OF OUR COUNT ALGORITHM

In the remainder of this paper, this section elaborates our key ideas for our Count algorithm. Section 4 and 5 present the details and formal guarantees of our Count algorithm. Section 6 gives our algorithms for all the other problems.

Recall from Section 1 the overall approach of our Count algorithm, where the values move to the sinks by doing random walks, and later the sinks disseminate whatever they have collected. The following elaborates how we make this approach efficient and how we overcome the difficulty of unknown N, in the context of our Count algorithm.

3.1 ρ-Bounded-probability Random Walk

Recall that each node is chosen as a sink with about $\frac{\log N}{N^{1/7}}$ probability. (The algorithm does not know N — we deal with that later.) For a random walk to reach some sink with high probability, it suffices for the walk to visit about $N^{\frac{1}{7}}$ distinct nodes. Researchers have so far mainly considered two types of random walks in dynamic networks: *simple random walks* [5–8, 13, 29] and *max-degree random walks* [3, 7, 8, 14]. At each node, a *simple random walk* simply follows each incidental edge with the same probability. In comparison, a *max-degree random walk* moves to each of the node's neighbors with probability $\frac{1}{\gamma_{max}}$, and remains at the node with the remaining probability. Here γ_{max} is the maximum degree of all nodes in the network. When $\gamma_{max} = N$, the max-degree random walk is also called the *lazy random walk*. The cover time of a simple random walk in a dynamic network can be exponential [7]. Hence it is unclear whether it can visit $N^{\frac{1}{7}}$ distinct nodes in polynomial, let alone sublinear, number of steps. For max-degree random walk, since we aim for a general result that does not depend on node degrees, we will have to use $\gamma_{max} = N$. In such a case, it already takes roughly $\Theta(N)$ steps for the walk to get out of a degree-1 node, making a sublinear result impossible. But max-degree random walk does give us hope: If the whole network had only $N^{\frac{1}{7}}$ nodes instead of N nodes, then it is known [14] that a max-degree random walk would in expectation cover all $N^{\frac{1}{7}}$ nodes (and hence visit $N^{\frac{1}{7}}$ distinct nodes) in $O(N^{\frac{3}{7}} \log N)$ steps, which is sublinear.

To enable random walks to visit $N^{\frac{1}{7}}$ distinct nodes *in a network with N nodes* within sublinear number of steps, we use a simple generalization of max-degree random walks, which we call *ρ-bounded-probability random walk* (or *ρ-BRW*), with ρ being a tunable parameter. At each node, a ρ-BRW goes to each of the node's neighbors with $\min(\frac{1}{\gamma}, \rho)$ probability, and remains at the current node with the remaining probability. Here γ is the current node's degree. Simple random walk is essentially ρ-BRW with $\rho = 1$, while max-degree random walk is ρ-BRW with $\rho = \frac{1}{\gamma_{max}}$. We later show that a ρ-BRW takes roughly $O(\frac{1}{\rho^3} \log N) \cdot \frac{1}{\rho}$ steps to visit $\frac{1}{\rho}$ distinct nodes, or roughly $O(N^{\frac{4}{7}} \log N)$ steps to visit $N^{\frac{1}{7}}$ distinct nodes.

3.2 Implementing the Random Walks in Dynamic Networks

Recall from Section 1 that implementing random walks in dynamic networks is difficult: When a node u directs the walk to node v, node v may no longer be u's neighbor anymore. The following explains how we overcome this.

Allow random walk steps to fail. We use 4 rounds to do one step in ρ-BRW, while leveraging the condition that $T \geq 4$. This is also the only part in our entire design that requires the condition of $T \geq 4$. Let us consider round 1 through 4 as an example. For $1 \leq i \leq 4$, define G_i to be the maximum (in terms of the number of edges) common subgraph that is contained in all topologies from round 1 through round i. Obviously, G_i is a subgraph of G_{i-1}, and all the G_i's are connected, for T-interval dynamic networks with $T \geq 4$.

Let us first focus on rounds 3 and 4. We will let each node send some message in each round. At the end of round 3, a node u examines the set $W(u)$ of its neighbors in G_3, after it has received messages from them. In round 4, for the purpose of doing ρ-BRW, we let node u blindly assume that its neighbors are still $W(u)$: Namely, node u chooses each node w in $W(u)$ with $\min(\frac{1}{|W(u)|}, \rho)$ probability, and then sends a message containing the id of w, to direct the random walk to w. If in round 4, node w is no longer u's neighbor, then the step fails, and the random walk will simply stay at node u. (Node u will know this, based on whether it receives any message from w in round 4.)

It is not immediately clear what kind of random walk this design gives us. If $|W(u)| \leq \frac{1}{\rho}$, then the above implementation precisely match how ρ-BRW would behave on the topology G_4. The reason is that the walk goes to each of u's neighbor in G_4 with exactly ρ probability (recall that G_4 is a subgraph of G_3), and stays at u with the remaining probability. Since G_4 is connected when $T \geq 4$, we are now essentially doing ρ-BRW over a (new) 1-interval dynamic network, where the topologies in this dynamic network are all the G_4's. This new dynamic network is different from the original T-interval dynamic network. But the properties of ρ-BRW still hold in this new dynamic network.

If $|W(u)| > \frac{1}{\rho}$, then the algorithm will deviate from the behavior of ρ-BRW. For example, imagine that $\rho = 0.5$, and that u has 100 neighbors in G_3 and 50 neighbors in G_4. A ρ-BRW on G_4 should go to each of u's neighbors in G_4 with 0.02 probability. But our above design will direct the walk to each of u's neighbors in G_4 with only 0.01 probability.

Guiding toward sinks. We circumvent the above problem in the following way. Recall the goal of our random walk is to reach some sink. If $|W(u)| > \frac{1}{\rho}$ and hence is large, then $W(u)$ likely contains some sinks. Furthermore by the definition of G_3, all these sinks must have been u's neighbors in G_1, G_2, and G_3. Thus u actually had some good opportunity in G_1, G_2, and G_3 to directly make the walk go to one of its sink neighbors, and does not need to worry whether in G_4 it can properly do ρ-BRW. Specifically, in each round $i \in [2, 3]$, node u picks a random sink neighbor w that it had in G_{i-1}, and directs the random walk to w. If w is indeed still u's neighbor in round i, then the random walk proceeds to w. Otherwise the walk stays at u.

To see why this works, let S_i be the set of sink neighbors that u has in G_i. We obviously have $|S_1| \geq |S_2| \geq |S_3|$. If each node is chosen as a sink with probability q, then we have $E[|S_3|] = |W(u)| \cdot q > \frac{1}{\rho} \cdot q$ and $E[|S_1|] \leq N \cdot q$. Hence the decrease from $E[|S_1|]$ to $E[|S_3|]$ is at most a factor of $N\rho$. This implies that either $\frac{E[|S_2|]}{E[|S_1|]} \geq 1/\sqrt{N\rho}$ or $\frac{|E[S_3]|}{|E[S_2]|} \geq 1/\sqrt{N\rho}$. In the former case, round 2 gives node u a non-trivial success probability of $1/\sqrt{N\rho}$, for its attempt at directing the walk to some sink. In the latter case, round 3 provides such an opportunity. This is also the reason why we use 3 rounds (i.e., round 1 through 3) for guiding the walk toward sinks — otherwise we would not get a success probability of $1/\sqrt{N\rho}$.

To summarize, we either will faithfully follow the behavior of ρ-BRW, or will have some non-trivial probability of successfully directing the walk to a sink. Later, we will view the walk as consisting of many *segments*, with each segment having such property, so that this non-trivial probability builds up.

3.3 Checking Soundness Using Random Shares

Recall that for the final count to be correct, we need the 1.0 value from each node to successfully reach some sink, and we need all the sinks to successfully disseminate their respective sums (of the values collected) to all nodes. This may not happen if: i) there are too few sinks, ii) some random walk is too short, iii) the parameter ρ in ρ-BRW is too large/small, or iv) some node outputs before hearing from all sinks. The central difficulty here is that the algorithm does not know N and D — otherwise we could easily determine, for example, the parameter ρ or how long a node should wait before outputting.

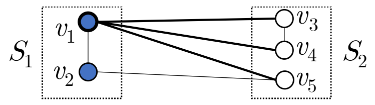

Figure 1: The topology of the dynamic network in the share distribution/replication round.

Our solution. Instead of trying to resolve the above 4 problems individually, we propose a novel and elegant mechanism to enable a node to directly check the soundness of its final count. Specifically, each node maintains rough guesses on N and D, and uses those to determine the parameters for the above 4 places. Using our mechanism, a node w can efficiently tell whether it has seen values from all nodes (indirectly via the sinks). If so, w can safely output the final count. Otherwise, w will re-try using larger (e.g., by doubling) guesses of N and D.

To provide some intuition of our mechanism, imagine that each node has a random number from the group $\mathbb{Z}_p$, where p is a large prime number. (Later we will see that this random number is obtained by summing up a set of random *shares*.) Further imagine that the random numbers from all the nodes add up to exactly 0 modulo p. Each node u then attaches its random number onto its original 1.0 value, to form a *tuple*. Instead of only dealing with the values, our Count algorithm now propagates the tuples, with proper aggregation when needed. (The values and the random numbers in the tuples are aggregated separately.) Eventually, a node w computes both the sum of the original 1.0 values and the sum of all the random numbers. We will ensure that if w has not added up all tuples from all nodes, the sum of the random numbers is unlikely to be 0. This then enables w to judge whether it has seen values from all nodes.

Obtaining the random numbers. It is not obvious how to obtain the random numbers as needed in the above: We want some correlation among these numbers so that all of them add up to 0, and yet we do not want excessive correlation which may cause some subset to also add up to 0. We use the following elegant way to achieve this (but with a caveat), in just a single round. In this single round, each node u first chooses a uniformly random x from $\mathbb{Z}_p$ as its *share*. Next, node u sends a copy of its share to all its neighbors, using a single message containing x. Let k be the number of u's neighbors in this round. (At the end of this round, node u will know the value of k, after u receives messages from its neighbors.) Let $y = p - x$ be the additive inverse of x in the group $\mathbb{Z}_p$ — namely, y is a *inverse share* for the share x. At the end of the round, node u adds up $k \cdot y$ and all the k shares that it receives from its k neighbors, to get its final random number σ_u.

It is not hard to see that $\sum_u \sigma_u = 0$, since each share has a corresponding inverse share in the system. But all these σ_u's can be correlated in complex ways, and it is not obvious whether some subsets of them may also add up to 0. Consider the example in Figure 1, and any given node w (not shown in the figure). Imagine that w has added up all the tuples from all nodes in S_1 in the figure, and has not seen any tuple from nodes in S_2. We hope that $\sum_{u \in S_1} \sigma_u \neq 0$. Let the random shares chosen by v_1 through v_5 be x_1 through x_5, respectively. Note that for any edge between two nodes in S_1, the share sent via that edge and the corresponding inverse share kept by the share's sender always add up to 0. Hence we have $\sum_{u \in S_1} \sigma_u = 3(p - x_1) + (p - x_2) + x_3 + x_4 + 2x_5$. To reason about $\Pr[\sum_{u \in S_1} \sigma_u = 0]$, we imagine that all random shares except x_1 have been fixed. We then have $\Pr[\sum_{u \in S_1} \sigma_u = 0] = \Pr[3(p - x_1)$ is the inverse of $(p - x_2) + x_3 + x_4 + 2x_5]$. Since p is a prime, as long as 3 is not a multiple of p, this probability is just $\frac{1}{p}$. (Recall that here "3" is the number of neighbors of v_1 that are in S_2.) This in turn implies that node w outputs a wrong result only with probability $\frac{1}{p}$.

As a caveat for the above reasoning to hold, S_1 needs to have some node v_1, such that the number of v_1's neighbors in S_2 is not a multiple of p. Later we will simply make p larger than the degree of all the nodes: Specifically, every node checks whether its degree is at least as large as the current p, and if yes, it raises a flag to force p to increase.

3.4 Challenges in Choosing the Prime Number p

The above error probability of $\frac{1}{p}$ is for a given node w. Ideally we want p to be at least $\frac{N}{\epsilon}$, so that the algorithm can get ϵ error probability via a union bound across the N nodes. But we do not know N, and hence cannot easily set p. To overcome this issue, our first idea is to try to set $p \geq \frac{A}{\epsilon}$, where A is the largest id among all the N nodes. Note that here we are viewing the id A as an integer value. Since all the N ids are unique and since the size of each id is $O(\log n)$, we must have $N \leq A = \Theta(\text{poly}(N))$. Furthermore, Bertrand's postulate [28] shows the existence of some prime number in $[\frac{A}{\epsilon}, \frac{2A}{\epsilon}]$. Hence a p value close to $\frac{A}{\epsilon}$ is sufficiently large to bound the final error probability to be within ϵ, and is also sufficiently small for a message to use only $O(\log N)$ bits to encode any element in $\mathbb{Z}_p$.

But the nodes do not know A either. In fact, finding the largest id is at least as hard as LeaderElection, which is one of the problems we intend to solve. To resolve this, in each round in the background, we let every node send the largest id it has seen so far, initially its own id. In any given round, let a_w be the largest id that node w has seen so far in the execution. Each node w can then use some p value (denoted as p_w) such that $p_w \geq \frac{a_w}{\epsilon}$. Note that different nodes may use different p values, because the largest id they have seen so far may be different. Conceptually, we now further include the p value in each tuple. Whenever a node sees a p value different from its own, it will raise a flag, and flood the flag to all nodes. If node w does not see any flag, and furthermore sees the sum of all the random numbers in the tuples being 0, then the soundness checking passes on w.

While a_w will eventually grow to A, unfortunately, we may still experience excessive error probability before that happens. For example, imagine a network where for each $i \in [1, \frac{N}{1000}]$, there is a node with id $1000i$ that is surrounded by 999 nodes with ids smaller than $1000i$. If we set p_w to be close to $\frac{a_w}{\epsilon}$, then during the initial stage of the execution, a union bound across these $\frac{N}{1000}$ "locally-maximal" nodes will give an error probability of roughly $\sum_{i=1}^{N/1000} \frac{\epsilon}{1000i} = \Theta(\epsilon \log N)$, which exceeds ϵ. (We cannot simply increase p_w by a factor of $\log N$, since we do not know N.) To avoid this problem, we use a simple trick: We make $p_w \geq \frac{4a_w^2}{\epsilon}$, instead of just $\frac{a_w}{\epsilon}$. Since the smallest ids the nodes can have are 1 through N, and since each node initially always sees its own id, the error probability (after a union bound) is now at most $\sum_{i=1}^{N} \frac{\epsilon}{4i^2} < \epsilon$.

3.5 Terminate after Outputting

With our design so far, some node u (e.g., if u is close to all the sinks) may pass the soundness checking much earlier than some other node w, causing u to output the final count much earlier than w. If node u immediately terminates upon outputting, then node w may not be able to compute the correct count any more. In fact, the early termination of all such u's could even partition the network. Note that we cannot simply ask a node to count the number of nodes that have already output, and then terminate after all nodes have output. The reason is that during this second counting process, again, node u may complete the counting faster and then terminate, before node w completes the counting. This again causes problem for w.

Overview of our idea. We propose a novel idea of *dual-schedule termination* to overcome the above challenge, and to enable the nodes to properly and efficiently terminate, as following. When a node u outputs N in a certain round t, node u will immediately schedule a *slow termination* for all nodes in the system, where the slow termination will happen in round $t_1 = t + N$. Node u does this by flooding a TERM1 message containing the values N and t_1. All other nodes will keep relaying this TERM1 message in every round forever. A node receiving this TERM1 message will terminate in round t_1, if it has not already terminated by then. (Since the TERM1 message contains N as well, a node will also output N upon receiving the message, if it has not yet output. Future outputs on this node will be suppressed/discarded.) Note that it is possible for multiple u's to do such scheduling concurrently, and for multiple TERM1 messages to propagate in the network. A node only processes and forwards the TERM1 message with the smallest t_1 that it has ever seen, and ignore all other TERM1 messages.

Since the dynamic diameter never exceeds N, one can then see that no node will ever terminate, before all nodes have received some TERM1 message and have output N. This in turn means that one node's termination will not negatively affect other nodes. But this slow termination adds an additive $O(N)$ term to the time complexity, which is undesirable.

To achieve sublinear complexity, we schedule another *fast termination.* Such scheduling will be done after the system has run the Count algorithm (together with the above slow termination mechanism) for sometime. To schedule the fast termination, we will count the number M of nodes that have already seen some TERM1 message. The computation of M will be done by a second and independent invocation of our Count algorithm, with a trivial modification to be described later. The scheduling of fast termination will be done only if $M = N$. If $M < N$, the system will simply retry later.

Now if a node u observes $M = N$ in round t', it will schedule a *fast termination* for all nodes in the system, where the fast termination will happen in round $t_2 = t' + N^{\frac{1}{7}}$. This is done by node u flooding a TERM2 message containing the value t_2, and all other nodes will keep relaying this TERM2 message in every round forever. Again, multiple u's may do such scheduling concurrently, and multiple TERM2 messages may propagate in the network. A node only processes and forwards the TERM2 message with the smallest t_2 that it has ever seen, and ignore all other TERM2 messages. Finally, a node that has received a TERM2 message will terminate in round t_2 (as indicated in that message), if it has not yet already terminated by then.

Effect on correctness. It is not immediately clear, in the above design, how one node's termination may affect other nodes. To facilitate discussion, we say that a node terminates *according to TERM1*, if it terminates in some round t_1 as indicated by some TERM1 message. We similarly define the notion of a node terminating *according to TERM2.* A node may receive both a TERM1 and a TERM2 message, in which case the node will terminate either according to TERM1 or TERM2, depending on whether t_1 or t_2 is smaller. (Both cases are possible.)

In our design, a node w may invoke our Count algorithm to compute N or M. While it is doing so, some other nodes may have already terminated. The following explains why such termination will not cause any problem. Our reasoning will rely on a central invariant: When any node u terminates (either according to TERM1 or TERM2), all nodes must have output N and must have received some TERM1 message.

We first consider the case where node w is still running our Count algorithm to compute N, when some other node u has already terminated. If u has terminated according to TERM1, then by our design, node w must have already seen a TERM1 message, and furthermore must have output N already upon receiving that TERM1 message. Hence w's invocation of our Count algorithm for computing N is actually no longer needed, and the output from that invocation will be discarded anyway. Next, if u has terminated according to TERM2, then by our design, we have $M = N$. This again means that node w must have already seen a TERM1 message and have output N.

The second case is where node w is still running our Count algorithm to compute M, when some other node u has already terminated. By similar reasoning, when u terminates, all nodes (including w) must have already seen some TERM1 message. This means we have $M = N$. Now even if w's computation of M is wrong (due to u's termination) and even if w incorrectly thinks $M < N$, the only effect is that w will not schedule a fast termination. But all nodes will at least terminate according to TERM1 already. Not scheduling a fast termination will then only affect time complexity and not correctness.

Effect on time complexity. We now explain why our termination design will not increase the asymptotic time complexity. It is easy to see that the slow termination, as scheduled by TERM1 messages, is guaranteed to happen within N rounds after the nodes have computed N. If D is large, or more specifically if $D \geq N^{\frac{1}{7}}$, this slow termination is already good enough, since the time complexity of the Count algorithm will already dominate this $O(N)$ term.

If $D < N^{\frac{1}{7}}$, then the fast termination (as scheduled by TERM2 messages) will help us to achieve the desired time complexity. Specifically, the computation of N and M will take $O(DN^{\frac{6}{7}}\text{polylog}(N))$ rounds, and it takes $O(D)$ rounds for a TERM1 message to reach all nodes. Hence within roughly $O(DN^{\frac{6}{7}}\text{polylog}(N)) + O(D) = O(DN^{\frac{6}{7}}\text{polylog}(N))$ rounds, some node u will finish computing both N and M, and will see $M = N$. Node u will then schedule a fast termination, by flooding a TERM2 message. All nodes will receive this TERM2 message within $N^{\frac{1}{7}}$ rounds, and will all terminate after at most $N^{\frac{1}{7}}$ rounds. This $N^{\frac{1}{7}}$ term will be dominated by the $O(DN^{\frac{6}{7}}\text{polylog}(N))$ time complexity of the Count algorithm. Putting it another way, when D is small, the fast termination will occur much earlier than the slow termination, which helps us to achieve the desired sublinear complexity.

4 OUR COUNT ALGORITHM

This section gives the pseudo-code/proofs for our Count algorithm. Recall from Section 3.3 that each node constructs a tuple, consisting of a value of 1.0 and a random number. Algorithm 1 gives the pseudo-code for the subroutine TuplesGoToSinks(), which gathers these tuples from all the nodes to all the sinks. Algorithm 2 is our Count algorithm, which uses TuplesGoToSinks(), together with another subroutine SinksFloodTuples(), to compute the final count. Here SinksFloodTuples() tries to flood the tuples gathered at the sinks to all nodes, and its pseudo-code is deferred to Appendix A.

Throughout these algorithms, each send operation corresponds to the advancement to the next round, ϵ is the target error probability, and we use constants $c = \frac{1}{7}$ and c' where c' is the constant from Theorem 4.1 later. We use A to denote the largest node id among all the N nodes. In Algorithm 2, each node u maintains its guesses for N, D, and A, which are denoted as n, d and a_u, respectively. These guesses are fed into TuplesGoToSinks() and SinksFloodTuples(). It will be clear later that at any point of time, all nodes must have the same guesses on N and D, and may have different guesses on A. Hence we put a subscript on a_u, but not on n and d.

4.1 Tuples Go To Sinks

Algorithm 1 lets each node become a sink with probability $\min(1, \frac{8}{n^c}(\ln\frac{1}{\epsilon} + \ln(64n^2) + \ln\ln\frac{16n}{\epsilon}))$, and then gathers all the tuples to the sinks, by trying to do a ρ-BRW with $\rho = \frac{1}{n^c}$ from each node. Each ρ-BRW has $(\lceil 16\sqrt{n^{1-c}}\ln\frac{16n}{\epsilon}\rceil) \times (\lceil 2c'n^{3c}\ln n^c\rceil + 1)$ *steps*. Line 3 through 24 implements a single step, using 4 rounds while following the idea in Section 3.2. At Line 16, W_u is the set of u's neighbors that have always been u's neighbors in the past 3 rounds. If $|W_u| \leq n^c$, at Line 18 the walk will try to continue to some neighbor of u. Otherwise, Line 4 through 15 should already have had some non-trivial probability of successfully guiding the walk to some sink. Each node maintains a flag at Line 27 and 28, to indicate whether the prime number used by itself is the same as other nodes, as explained in Section 3.3.

Theorem 4.2 next proves the property of ρ-BRW: With $\frac{1}{2}$ probability, a ρ-BRW visits a new node every $O(\frac{1}{\rho^3}\log\frac{1}{\rho})$ steps, if it has visited less than $\frac{1}{\rho}$ distinct nodes so far. The proof of Theorem 4.2 invokes the following existing result on lazy random walk [14] (which is the same as $\frac{1}{N}$-BRW):

THEOREM 4.1 (ADAPTED FROM THEOREM 2 IN [14]). *There exists constant $c' > 0$, such that for all T-interval dynamic networks with N nodes (for all T and N), the cover time of $\frac{1}{N}$-BRW is at most $c'N^3\ln N$.*

Algorithm 1 TuplesGoToSinks $(n, p_u, \texttt{flag}_u, \texttt{tuple}_u)$ for node u

```
1:  $\texttt{isSink}_u \leftarrow$ True with probability $\min(1, \frac{8}{n^c}(\ln\frac{1}{\epsilon} + \ln(64n^2) + \ln\ln\frac{16n}{\epsilon}))$, and $\texttt{isSink}_u \leftarrow$ False with remaining probability;
2:  **repeat** $(\lceil 16\sqrt{n^{1-c}}\ln\frac{16n}{\epsilon}\rceil) \times (\lceil 2c'n^{3c}\ln n^c\rceil + 1)$ **times**
3:      **send** $\langle$SINK, isSink$\rangle$; let $S_u$ be the set of nodes from which I receive $\langle$SINK, True$\rangle$;
4:      **repeat** 2 **times** // The following loop takes 2 rounds.
5:          **if** ($S_u \neq \emptyset$) and ($\texttt{isSink}_u$ = False) **then**
6:              choose a node $v$ uniformly randomly from $S_u$;
7:              **send** message $\langle$MOVE, $v$, $\texttt{tuple}_u\rangle$;
8:              **if** I received some message from $v$ in this round **then** $\texttt{tuple}_u \leftarrow \langle 0, 0\rangle$;
9:          **else**
10:             **send** an empty message;
11:         **end if**
12:         for each $\langle$MOVE, $u$, $\texttt{tuple}'\rangle$ message received in this round: $\texttt{tuple}_u \leftarrow \texttt{tuple}_u + \texttt{tuple}'$;
13:         $S_u \leftarrow S_u \cap \{v \mid$ I have received some messages from node $v$ in
14: this round$\}$;
15:     **end**
        // Line 16 to 24 takes one round.
16:     $W_u \leftarrow \{v \mid$ I have received some message from node $v$ in each of the previous 3 rounds$\}$;
17:     **if** ($|W_u| \leq n^c$) and (isSink = False) **then**
18:         choose a node $v$ such that each node in $W_u$ is chosen with probability $\frac{1}{n^c}$, and set $v$ to be $u$ with the remaining $1 - |W_u|\frac{1}{n^c}$ probability;
19:         **send** message $\langle$MOVE, $v$, $\texttt{tuple}_u\rangle$;
20:         **if** I receive any message from node $v$ in this round **then** $\texttt{tuple}_u \leftarrow \langle 0, 0\rangle$;
21:     **else**
22:         **send** an empty message;
23:     **end if**
24:     for each $\langle$MOVE, $u$, $\texttt{tuple}'\rangle$ message received in this round: $\texttt{tuple}_u \leftarrow \texttt{tuple}_u + \texttt{tuple}'$;
25: **end**
26: **return** $\langle\texttt{flag}_u, \texttt{tuple}_u\rangle$;

27: **Concurrently** with the above, each node does the following in every round:
28:     **send** $\langle$FLAG, $\texttt{flag}_u$, $p_u\rangle$; **if** (I receive message $\langle$FLAG, $\texttt{flag}'$, $p'\rangle$ with either $\texttt{flag}'$ = True or $p' \neq p_u$ in this round) **then** $\texttt{flag}_u \leftarrow$ True;
```

THEOREM 4.2. *Let c' be the constant from Theorem 4.1. Consider any $T \geq 1$, $N \geq 1$, and any T-interval dynamic network H with N nodes. For any given ρ where $\frac{1}{\rho}$ is an integer, and any given set B of nodes where $|B| < \min(\frac{1}{\rho}, N)$, if we do a ρ-BRW on H starting from any node and for $2c'\frac{1}{\rho^3}\ln\frac{1}{\rho}$ steps, then with probability at least $\frac{1}{2}$, that ρ-BRW visits some node not in B.*

PROOF. See Appendix B. □

Using Theorem 4.2, Lemma 4.4 next shows that with high probability, every tuple will reach some sink in Algorithm 1. Its proof follows the intuition in Section 3.2, and takes into account that Algorithm 1 sometimes deviates from ρ-BRW. To be rigorous, we will define the notion of a *path* that is traversed by a tuple, as specified by the pseudo-code. This path is different from a ρ-BRW, and our proof will establish connections between them.

Definition 4.3 (path). Consider any given execution of Algorithm 1, and any given node u. The *path* traversed by the tuple from u is a sequence of nodes $\psi = \psi_1\psi_2\psi_3\ldots$. For convenience, define $\psi_0 = u$. For $i \geq 1$, ψ_i is defined to be v if in round i, node v is a neighbor of node ψ_{i-1} and node ψ_{i-1} sends a message $\langle$MOVE, v, $\cdot\rangle$ at Line 7 or 19 of Algorithm 1. Otherwise ψ_i is defined to be ψ_{i-1}. We also write the last node in ψ as $\psi_\perp$.

LEMMA 4.4. *For any $T \geq 4$ and $N \geq 1$, consider any execution of Algorithm 1 with $n \in [N, 2N]$ and with all N nodes invoking Algorithm 1. For any node u and the corresponding path ψ of its tuple, with probability at least $1 - \frac{\epsilon}{4N}$, node $\psi_\perp$ is a sink.*

PROOF. See Appendix C. □

4.2 Computing Count

Algorithm 2 gives our final Count algorithm. Following Section 3.4, the algorithm maintains n and d, as rough guesses for N and D. For each pair of (n, d) values, the algorithm computes the count using TuplesGoToSinks() and SinksFloodTuples(), and the invokes the soundness checking mechanism to check whether the count is correct (Line 46). When soundness checking fails, Algorithm 2 will try larger (n, d) values. Obviously, we want to properly bound the total time complexity incurred by all these *trials*, where each trial corresponds to one specific (n, d) pair (i.e., Line 33 through 47). To do so, the algorithm maintains a certain budget on the number of rounds (i.e., the term 2^i at Line 32), and the budget doubles when

Algorithm 2 Count() for node u.

```
28: Define $f(d,n) = 100(c'+1)dn^{1-c}\log^2(4n)\log\frac{4}{\epsilon}$, which is an upper bound on the number of rounds taken by one iteration of the while loop from Line 33 to 47. Here the log is base 2.
29: $j \leftarrow 1$; $\texttt{result}_u \leftarrow \bot$;                    ▷ $j$ is the number of trials.
30: for $i \leftarrow 1,2,3,\cdots$ do
31:     $n \leftarrow 1$;
32:     while $f(1,n) \le 2^i$ do
33:         let $d$ be the largest integer such that $f(d,n) \le 2^i$;
34:         let $p_u$ be smallest prime number such that $p_u \ge \frac{4a_u^2}{\epsilon} \times (2j^2)$, where $a_u$ is the largest id I have seen so far;
            /* Line 35 to 39 is for share replication/distribution, and take one round. */
35:         $\texttt{flag}_u$ ←False; $\texttt{rand\_num}_u \leftarrow 0$; $x_u \leftarrow$ integer chosen uniform randomly from $[0, p_u - 1]$;
36:         send ⟨SHARE, $p_u, x_u$⟩;
37:         for each message in the form of ⟨SHARE, $p', x'$⟩ received in this round:
38:             if $p' \neq p_u$ then $\texttt{flag}_u \leftarrow$ True; else $\texttt{rand\_num}_u \leftarrow \texttt{rand\_num}_u + x' + (p_u - x_u)$;
39:         if I received messages from at least $p_u$ nodes in this round then $\texttt{flag}_u \leftarrow$ True;
40:         if I receive ⟨FIN, $\texttt{result}'$⟩ with $\texttt{result}' \neq \bot$ in this round then { $\texttt{result}_u \leftarrow \texttt{result}'$; output $\texttt{result}_u$; goto Line 50;}
            /* Line 41 to 46 compute the sum of the values and sum of the random numbers.*/
41:         $\texttt{tuple}_u \leftarrow \langle 1, \texttt{rand\_num}_u \rangle$;
42:         $\langle \texttt{flag}_u, \texttt{sinkTuple}_u \rangle \leftarrow$ TuplesGoToSinks($n, p_u, \texttt{flag}_u, \texttt{tuple}_u$);
43:         // see Algorithm 1 for the pseudo-code of TuplesGoToSinks()
44:         $\langle \texttt{flag}_u, \texttt{allTuple}_u \rangle \leftarrow$ SinksFloodTuples($n, p_u, \texttt{flag}_u, \texttt{sinkTuple}_u, d$);
45:         // see Appendix A for the pseudo-code of SinksFloodTuples()
46:         if ($\texttt{flag}_u$ = False) and ($\texttt{allTuple}_u.\texttt{count} > 0$) and ($\texttt{allTuple}_u.\texttt{rand\_num} = 0 \mod p_u$) then { $\texttt{result}_u \leftarrow \texttt{allTuple}_u.\texttt{count}$;
    output $\texttt{result}_u$; goto Line 50; }
47:         $n \leftarrow 2n$; $j \leftarrow j+1$;
48:     end while
49: end for
50: keep sending ⟨FIN, $\texttt{result}_u$⟩ in each round;

51: Concurrently with the code above, each node does the following in every round:
52:     send ⟨FIN, $\texttt{result}_u$⟩ and the largest id that I have seen so far (initially my own id);
53:     update the largest id I see using the received messages;
54:     if (I receive ⟨FIN, $\texttt{result}'$⟩ with $\texttt{result}' \neq \bot$) then $\texttt{result}_u \leftarrow \texttt{result}'$;
```

soundness checking fails. For each budget value, the algorithm tries all (n,d) pairs where n is a power of 2 and where d is the largest corresponding d such that the total number of rounds needed by that trial does not exceed the budget. The sequence of (n,d) pairs that each node tries will be exactly the same. Hence the nodes will start any given trial (for a give (n,d) pair) in the same round. Finally, we also need to properly bound the total error incurred in all these trials. To do so, we allow the j-th trial to incur an error probability of $\frac{\epsilon}{2j^2}$, and set p accordingly at Line 34. The resulting total error will be at most $\sum_{j=1}^{\infty}\frac{\epsilon}{2j^2} < \epsilon$. The following is the final theorem on our Count algorithm:

Theorem 4.5. *For all given $T \ge 4$, $N \ge 1$, and $\epsilon = \Omega(\frac{1}{\text{poly}(N)})$, with probability at least $1-\epsilon$, Algorithm 2 outputs N on all nodes within $O(DN^{\frac{6}{7}}\text{polylog}(N))$ rounds.*

Proof. See Appendix D. □

5 MAKING THE ALGORITHM TERMINATE

Algorithm 2 in the previous section focuses only on nodes *outputting N*. In many cases, it is further desirable for the nodes to *terminate* (i.e., stop execution), after outputting N. This section explains how to achieve this, without increasing the asymptotic time complexity in Theorem 4.5. Our design here follows the intuition in Section 3.5.

Flooding of TERM1 and TERM2 messages. When a node outputs N in round t at Line 46 in Algorithm 2, it will separately flood a message ⟨TERM1, N, t_1⟩, where $t_1 = t + N$, to all nodes. (We also view the node itself as receiving this message in the same round as it initiates the flooding.) A node receiving this TERM1 message will i) immediately output N as the result of the count, ii) keep sending/relaying this TERM1 message in each round, and iii) terminate in round t_1 if it has not terminated by round t_1.

With this mechanism, a node may potentially output the result of the count multiple times (including the output originally generated at Line 46 in Algorithm 2). Only the first output will materialize, while all later outputs will be suppressed. A node may also receive multiple TERM1 messages during its execution. The node only processes the TERM1 message with the smallest t_1 that it has ever seen, and ignore all other TERM1 messages.

After running Algorithm 2 (together with the above mechanism) for some time, we will count the number M of nodes that have received a TERM1 message by now. The computation of M will be done by a second and independent invocation of Algorithm 2, with a trivial modification. Details on this will be provided later. After a node computes M in round t', and if it observes $M = N$, then that node will flood a message $\langle \text{TERM2}, t_2 \rangle$ with $t_2 = t' + N^{\frac{1}{7}}$, to all nodes. A node receiving this TERM2 message will i) keep sending/relaying this TERM2 message in each round, and ii) terminate in round t_2 if it has not terminated by round t_2. Similar as before, if a node receives multiple such TERM2 messages, it only processes the one with the smallest t_2.

How and when to compute M. We first explain how to modify Algorithm 2 to count the number M of nodes that have received a TERM1 message. Recall that in Algorithm 2, each node contributes a value of 1 at Line 41. For computing M, a node simply contributes 1 if it has received a TERM1 message, otherwise it contributes 0.

So far we have not yet specified exactly when the nodes should start computing M. Ideally, we would compute M after some node u has finished computing N and after u's TERM1 message have reached all nodes. But we do not know when that will happen. Hence we will repeatedly compute M at multiple time points. Specifically, we proceed in *epochs*, where the i-th epoch has exactly 2^i rounds, for $i \geq 1$. In each epoch, the nodes invoke Algorithm 2 to count the number M of nodes that have received a TERM1 message by the beginning of that epoch. All these invocations are independent of, and are in parallel with, the invocation of Algorithm 2 for counting the total number N of nodes. The sequential invocations for computing M in different epochs are completely independent as well. An invocation for computing M in an epoch will be stopped at the end of that epoch, regardless of whether the computation of M has completed. This makes room for the invocation in the next epoch.

If a node u has already computed N and if in an epoch, the corresponding invocation of Algorithm 2 generates a result $M = N$ on u, then node u will immediately flood a TERM2 message as discussed earlier. Note that the flooding of TERM1 and TERM2 messages is global, and is not confined to be within an epoch. Hence a flooding initiated in one epoch will continue throughout all future epochs.

Finally, since we are invoking Algorithm 2 many times, we need to properly bound the overall error probability. Let ϵ be the desired final error probability. For the invocation of Algorithm 2 for computing N, we allow it to have an error probability of $\epsilon'_0 = \frac{\epsilon}{2}$. For the invocation of Algorithm 2 for computing M in epoch i, we allow it to have an error probability of $\epsilon'_i = \frac{\epsilon}{2^{i+1}}$.

Final guarantees. The following final theorem summarizes the formal guarantees of the above design. The proof is based on the intuitions in Section 3.5.

THEOREM 5.1. *For all $T \geq 4$, $N \geq 1$, and $\epsilon = \Omega(\frac{1}{\text{poly}(N)})$, with probability at least $1-\epsilon$, the algorithm described in Section 5 outputs N and also terminates on all nodes, within $O(DN^{\frac{6}{7}}\text{polylog}(N))$ rounds.*

PROOF. See Appendix E. □

6 OUR $O(DN^{\frac{6}{7}}\text{polylog}(N))$ ALGORITHMS FOR OTHER PROBLEMS

Our algorithmic framework for the Count algorithm can be applied to solving a number of other problems, all in $O(DN^{\frac{6}{7}}\text{polylog}(N))$ time complexity. (See Section 1 for the definitions of those problems.) To solve Sum, we simply tweak the Count algorithm so that each node uses its input (instead of using 1.0) as part of the tuple. To solve Max, recall that in the Count algorithm, we add up the values in the tuples when we aggregate multiple tuples. For solving Max, all we need is to instead take the maximum among of the values when aggregating. (For the random numbers in the tuples, during aggregation we still add them up.) The algorithm for solving Max, without any change, also solves Consensus. LeaderElection can be solved by having each node use its id as the input and then solving Max. Finally, to solve ConfirmedFlood, we insert an extra step between Line 40 and 41 in Algorithm 2. This extra step spends d rounds to let the designated node flood its input. A node u then sets $\texttt{flag}_u$ to be True, if it does not receive that input. When the designated node is about to output the final count in Algorithm 2 at Line 46, it will instead output "done", as needed in ConfirmedFlood. Other nodes will output nothing, when they are about to output. The proofs for the correctness/complexity of all these algorithms are almost identical to the proofs for the Count algorithm. Hence we do not repeat such proofs for clarity.

ACKNOWLEDGMENTS

This work is partly supported by the research grant MOE2017-T2-2-031 from Singapore Ministry of Education Academic Research Fund Tier-2.

REFERENCES

[1] Sebastian Abshoff, Markus Benter, Andreas Cord-Landwehr, Manuel Malatyali, and Friedhelm Meyer auf der Heide. 2013. Token Dissemination in Geometric Dynamic Networks. In *ALGOSENSORS.*

[2] M. Ahmadi and F. Kuhn. 2017. Multi-message Broadcast in Dynamic Radio Networks. In *ALGOSENSORS.*

[3] M. Ahmadi, F. Kuhn, S. Kutten, A. R. Molla, and G. Pandurangan. 2019. The Communication Cost of Information Spreading in Dynamic Networks. In *ICDCS.*

[4] J. Augustine, C. Avin, M. Liaee, G. Pandurangan, and R. Rajaraman. 2016. Information Spreading in Dynamic Networks Under Oblivious Adversaries. In *DISC.*

[5] John Augustine, Gopal Pandurangan, and Peter Robinson. 2013. Fast byzantine agreement in dynamic networks. In *PODC.*

[6] J. Augustine, G. Pandurangan, and P. Robinson. 2015. Fast Byzantine Leader Election in Dynamic Networks. In *DISC.*

[7] Chen Avin, Michal Koucky, and Zvi Lotker. 2008. How to explore a fast-changing world (cover time of a simple random walk on evolving graphs). In *ICALP.*

[8] C. Avin, M. Koucky, and Z. Lotker. 2018. Cover time and mixing time of random walks on dynamic graphs. *Random Structures & Algorithms* 52, 4 (2018), 576–596.

[9] P. Brandes and F. Meyer auf der Heide. 2012. Distributed Computing in Fault-prone Dynamic Networks. In *International Workshop on Theoretical Aspects of Dynamic Distributed Systems.*

[10] M. Chakraborty, A. Milani, and M. Mosteiro. 2018. A Faster Exact-Counting Protocol for Anonymous Dynamic Networks. *Algorithmica* 80, 11 (Nov 2018), 3023–3049.

[11] Bogdan S Chlebus, Dariusz R Kowalski, Jan Olkowski, and Jedrzej Olkowski. 2021. Consensus in Networks Prone to Link Failures. *arXiv preprint arXiv:2102.01251* (2021).

[12] Atish Das Sarma, Anisur Molla, and Gopal Pandurangan. 2012. Fast Distributed Computation in Dynamic Networks via Random Walks. In *DISC.*

[13] A. Das Sarma, A. Molla, and G. Pandurangan. 2015. Distributed Computation in Dynamic Networks via Random Walks. *Theor. Comput. Sci.* 581, C (May 2015), 45–66.

[14] O. Denysyuk and L. Rodrigues. 2014. Random Walks on Evolving Graphs with Recurring Topologies. In *DISC.*

[15] Mohsen Ghaffari, Nancy Lynch, and Calvin Newport. 2013. The cost of radio network broadcast for different models of unreliable links. In *PODC*.

[16] Bernhard Haeupler and David Karger. 2011. Faster information dissemination in dynamic networks via network coding. In *PODC*.

[17] Irvan Jahja and Haifeng Yu. 2020. Sublinear Algorithms in T-interval Dynamic Networks. In *SPAA*.

[18] J. Katz and Y. Lindell. 2020. *Introduction to modern cryptography.* CRC press.

[19] D. Kowalski and M. Mosteiro. 2018. Polynomial Counting in Anonymous Dynamic Networks with Applications to Anonymous Dynamic Algebraic Computations. In *ICALP*.

[20] Dariusz R Kowalski and Miguel A Mosteiro. 2019. Polynomial anonymous dynamic distributed computing without a unique leader. In *ICALP*.

[21] Dariusz R Kowalski and Miguel A Mosteiro. 2021. Supervised Average Consensus in Anonymous Dynamic Networks. In *SPAA*.

[22] Fabian Kuhn, Nancy Lynch, and Rotem Oshman. 2010. Distributed Computation in Dynamic Networks. In *STOC*.

[23] G. Luna, R. Baldoni, S. Bonomi, and I. Chatzigiannakis. 2014. Conscious and unconscious counting on anonymous dynamic networks. In *International Conference on Distributed Computing and Networking*.

[24] G. Luna, R. Baldoni, S. Bonomi, and I. Chatzigiannakis. 2014. Counting in Anonymous Dynamic Networks Under Worst-Case Adversary. In *IEEE International Conference on Distributed Computing Systems*.

[25] G. Luna, S. Bonomi, I. Chatzigiannakis, and R. Baldoni. 2013. Counting in anonymous dynamic networks: An experimental perspective. In *ALGOSENSORS*.

[26] O. Michail, I. Chatzigiannakis, and P. Spirakis. 2013. Naming and Counting in Anonymous Unknown Dynamic Networks. In *Proceedings of International Symposium on Stabilization, Safety, and Security of Distributed Systems*.

[27] D. Peleg. 1987. *Distributed Computing: A Locality-Sensitive Approach.* Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania.

[28] S. Ramanujan. 1919. A proof of Bertrand's postulate. *Journal of the Indian Mathematical Society* 11, 181-182 (1919), 27.

[29] T. Sauerwald and L. Zanetti. 2019. Random Walks on Dynamic Graphs: Mixing Times, Hitting Times, and Return Probabilities. In *ICALP*.

[30] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.

[31] H. Yu, Y. Zhao, and I. Jahja. 2018. The Cost of Unknown Diameter in Dynamic Networks. *J. ACM* 65, 5 (Sept. 2018), 31:1–31:34.

A PSEUDO-CODE FOR SINKSFLOODTUPLES()

Algorithm 3 gives the pseudo-code for the subroutine SinksFloodTuples(), which tries to flood all the tuples gathered at all the sinks to every node. Note that each sink will aggregate the tuples they have collected, and then flood only a single aggregated tuple. To simplify reasoning, the algorithm does not just let the sinks flood — instead, it lets all nodes that are holding tuples flood. (For example, a random walk carrying a tuple may eventually only reach some non-sink node, causing the non-sink node to hold a tuple.) Algorithm 3 let all such nodes flood their respective aggregated tuples, one by one and for up to $\lceil 2n\cdot\min(1, \frac{8}{n^c}(\ln\frac{1}{\epsilon}+\ln(64n^2)+\ln\ln\frac{16n}{\epsilon}))\rceil$ such nodes/aggregated tuples. If there are more than so many nodes holding tuples, then no all of them get chance to flood. Specifically, Line 59 to 62 uses d rounds to select the largest id among the nodes who hold aggregated tuples. The node with that id (assuming $d \geq D$) will then flood its tuple, over d rounds at Line 72 to 75, and will no longer participate in future selections. In each of the d rounds, each node in the network only participates in the d-round flooding of the tuple originated from the largest id it saw earlier.

When $d < D$, different nodes may disagree over which id is selected as the largest, multiple nodes may flood their tuples simultaneously, and some flooding may only be done partially since not all nodes participate. Nevertheless, there will be no creation of new tuples, and the only effect is that not all aggregated tuples are properly flooded. Our soundness checking mechanism in Algorithm 2 can properly detect such a case, and then force d to increase. Finally, similar to Algorithm 1, each node maintains a flag in Line 80 in Algorithm 3.

B PROOF FOR THEOREM 4.2

We restate and prove Theorem 4.2 below:

Theorem 4.2. *Let c' be the constant from Theorem 4.1. Consider any $T \geq 1$, $N \geq 1$, and any T-interval dynamic network H with N nodes. For any given ρ where $\frac{1}{\rho}$ is an integer, and any given set B of nodes where $|B| < \min(\frac{1}{\rho}, N)$, if we do a ρ-BRW on H starting from any node and for $2c'\frac{1}{\rho^3}\ln\frac{1}{\rho}$ steps, then with probability at least $\frac{1}{2}$, that ρ-BRW visits some node not in B.*

Proof. Let node u be the starting node of the random walk, and let $k = 2c'\frac{1}{\rho^3}\ln\frac{1}{\rho}$. We obviously only need to prove the case where $u \in B$. Let V be the set of all nodes. Let H_1 be the dynamic network whose round-t topology ($t \geq 1$) is the multi-graph derived from the round-t topology in H, by compressing all nodes in $V \setminus B$ into one special node α and by discarding all self loops. H_1 hence contains exactly $|B|+1$ nodes. Let E_1 be the event that a ρ-BRW of k steps in H starting from node u visits some node $v \notin B$, and E_2 be the event that a ρ-BRW of k steps starting from node u in H_1 visits α. It is easy to see that conditioned upon the random walking currently being on some node $v \in B$, the probability of moving to α in H_1 is the same as the probability of moving to some $w \in V \setminus B$ in H. Hence by a simple coupling argument, we have $\Pr[E_1] = \Pr[E_2]$.

Now we construct another dynamic network H_2 whose round-t topology ($t \geq 1$) is the (simple) graph derived from the round-t topology in H_1, by replacing all the multi-edges with a simple edge and by adding $\frac{1}{\rho} - |B| - 1$ degree-1 dummy nodes as neighbors

Algorithm 3 SinksFloodTuples(n, p_u, $\texttt{flag}_u$, $\texttt{tuple}_u$, d) for node u

```
allTuple_u ← ⟨0, 0⟩;
repeat $\lceil 2n \cdot \min(1, \frac{8}{n^c}(\ln\frac{1}{\epsilon} + \ln(64n^2) + \ln\ln\frac{16n}{\epsilon}))\rceil$ times
    // Find the largest id among all those nodes that have something to flood.
    if tuple_u ≠ ⟨0, 0⟩ then max_sink_id_u ← u; else max_sink_id_u ← 0;
    repeat d times
        send ⟨MAXID, max_sink_id_u⟩;
        for each message ⟨MAXID, m′⟩ received in this round: max_sink_id_u ← max(max_sink_id_u, m′);
    end

    // Check if I am being selected.
    if u = max_sink_id_u then
        flooding_u = ⟨FLOOD, u, tuple_u⟩; tuple_u ← ⟨0, 0⟩;
    else
        flooding_u = null;
    end if

    // Flood for d rounds.
    repeat d times
        if flooding_u ≠ null then send flooding_u; else send an empty message;
        if I receive ⟨FLOOD, max_sink_id_u, ·⟩ then set flooding_u to be that message;
    end
    if flooding_u ≠ null then allTuple_u ← allTuple_u + tuple′ where flooding_u = ⟨·, ·, tuple′⟩;
end
return ⟨flag_u, allTuple_u⟩;

Concurrently with the above, each node does the following in every round:
     send ⟨FLAG, flag_u, p_u⟩; if (I receive message ⟨FLAG, flag′, p′⟩ with either flag′ = True or p′ ≠ p_u in this round) then flag_u ←
    True;
```

of α in every round. Note that H_2 has exactly $\frac{1}{\rho}$ nodes. Let E_3 be the event that a ρ-BRW on H_2 of k steps starting from node u visits α. By Theorem 4.1 and with a Markov inequality, we have $\Pr[E_3] \geq \frac{1}{2}$. To relate E_3 to E_2, note that the probability of reaching α in each step of random walk in H_2 is no larger than in H_1. A coupling argument then gives $\Pr[E_2] \geq \Pr[E_3]$. Hence we have $\Pr[E_1] = \Pr[E_2] \geq \Pr[E_3] \geq \frac{1}{2}$, which completes the proof. □

C PROOF FOR LEMMA 4.4

To prove Lemma 4.4, we first prove another lemma, Lemma C.1. Define $\Gamma_{[t_1:t_2]}(u) = \{w \mid w \text{ is a neighbor of } u \text{ in round } t_1 \text{ through } t_2\}$, and $S_{[t_1:t_2]}(u) = \{w \mid w \text{ is a sink and } w \in \Gamma_{[t_1:t_2]}(u)\}$. Following the intuition in Section 3.2, Lemma C.1 next proves that under certain conditions, if $\Gamma_{[t_1:t_2]}(u)$ is large, then there exists some $t_1 \leq t < t_2$ such that $S_{[t_1:t+1]}(u)$ is not much smaller than $S_{[t_1:t]}(u)$. This will later help ensure a non-trivial probability of the algorithm successfully guiding the walk towards sinks. Define R_1, R_2, and R_3 to be the randomness used in Line 1, 6, and 18 in Algorithm 1, respectively.

Lemma C.1. *For any given $T \geq 4$, $N \geq 1$, and outcomes of randomness R_2 and R_3, consider any execution of Algorithm 1 with $n \in [N, 2N]$ and with all N nodes invoking Algorithm 1. For any given round t_1 during which Line 3 in Algorithm 1 is executed and any given node u where $|\Gamma_{[t_1:t_1+2]}(u)| > n^c$, with probability at least $1 - \frac{\epsilon}{32n^2\ln(\frac{16n}{\epsilon})}$ over R_1, there exists $t \in \{t_1+1, t_1+2\}$ such that $\frac{|S_{[t_1:t]}(u)|}{|S_{[t_1:t-1]}(u)|} \geq \frac{1}{2\sqrt{N^{1-c}}}$.*

Proof. In Algorithm 1, each node independently becomes a sink with probability $q = \min(1, \frac{8}{n^c}(\ln\frac{1}{\epsilon} + \ln(64n^2) + \ln\ln\frac{16n}{\epsilon}))$, with the probability being defined over R_1. If $q = 1$, then we must have $\frac{S_{[t_1:t_1+2]}(u)}{S_{[t_1:t_1]}(u)} > \frac{n^c}{N} > \frac{1}{4N^{1-c}}$. Hence there must exist $t \in \{t_1+1, t_1+2\}$ such that $\frac{|S_{[t_1:t]}(u)|}{|S_{[t_1:t-1]}(u)|} \geq \frac{1}{2\sqrt{N^{1-c}}}$. If $q < 1$, then by Chernoff bound, we have $\Pr[S_{[t_1:t_1+2]}(u) \leq 0.5n^cq] \leq \exp(-\frac{1}{8}n^cq) = \frac{\epsilon}{64n^2\ln(\frac{16n}{\epsilon})}$, and $\Pr[S_{[t_1:t_1]}(u) \geq 2Nq] \leq \exp(-\frac{1}{3}Nq) \leq \frac{\epsilon}{64n^2\ln(\frac{16n}{\epsilon})}$. This means that with probability at least $1 - \frac{\epsilon}{32n^2\ln(\frac{16n}{\epsilon})}$, we have $\frac{S_{[t_1:t_1+2]}(u)}{S_{[t_1:t_1]}(u)} \geq \frac{0.5n^cq}{2Nq} \geq \frac{1}{4N^{1-c}}$. Then, by definition, we have $S_{[t_1:t_1+2]}(u) \subseteq S_{[t_1:t_1+1]}(u) \subseteq S_{[t_1:t_1]}(u)$. Hence with probability at least $1 - \frac{\epsilon}{32n^2\ln(\frac{16n}{\epsilon})}$, there must exist $t \in \{t_1+1, t_1+2\}$ such that $\frac{|S_{[t_1:t]}(u)|}{|S_{[t_1:t-1]}(u)|} \geq \frac{1}{2\sqrt{N^{1-c}}}$. □

We restate and prove Lemma 4.4 below:

Lemma 4.4. *For any $T \geq 4$ and $N \geq 1$, consider any execution of Algorithm 1 with $n \in [N, 2N]$ and with all N nodes invoking Algorithm 1. For any node u and the corresponding path ψ of its tuple, with probability at least $1 - \frac{\epsilon}{4N}$, node $\psi_\perp$ is a sink.*

Proof. Let $h = 16\sqrt{n^{1-c}} \ln \frac{16n}{\epsilon}$ and $k = 2c'n^{3c} \ln n^c$, where c' is the constant from Theorem 4.1. For clarity, we prove for the case where n^c, $\frac{h}{4}$, and k are all integers. The case where they are not integers can be proved similarly.

Defining and analyzing ψ'. To facilitate proof, we first define another sequence of nodes $\psi' = \psi'_1\psi'_2\ldots$, and reason about its properties. Later we draw a connection between ψ' and ψ to prove the lemma. For any given outcomes of randomness R_1, R_2, and R_3, we define the corresponding ψ' to be the value that ψ would take, if the outcome of R_1 were replaced by some outcome such that there are no sinks elected among the N nodes. Intuitively, ψ' is the same as ψ except that ψ' ignores all the sinks. Note that while we define ψ' for given outcomes of randomness R_1, R_2, and R_3, the sequence ψ' actually does not depend on R_1 and R_2. The sequence ψ' has a length of $4h(k+1)$, and we divide ψ' into h *segments* of length $4(k+1)$ each. Eventually, we will define a notion of a segment being *nice*, and show that with high probability at least $\frac{h}{4}$ segments in ψ' are nice.

For any given segment starting from node ψ'_{t_0+1} for some t_0, we define a random variable $\eta = \eta_1\eta_2\ldots\eta_k$ such that $\eta_s = \psi'_{t_0+4s}$ for $1 \le s \le k$. Essentially, η is a subsequence of that segment. We next define a new dynamic network H_1 based on the original dynamic network H, where H_1 intuitively takes the maximum common subgraph across every 4 rounds in H, starting from round $t_0 + 5$. Specifically, H_1 has the same set of nodes as H, and for all $s \ge 1$, the round-s topology of H_1 is the maximum common subgraph of the topologies of H from rounds $(t_0 + 4s + 1)$ to $(t_0 + 4s + 4)$. Since H is a T-interval dynamic network with $T \ge 4$, the topology of H_1 in every round must be connected, and H_1 must be a 1-interval dynamic network. Define random variable $\phi = \phi_1\phi_2\ldots\phi_k$ to be the k-step $\frac{1}{n^c}$-BRW on H_1 that starts from node η_1, which implies that $\phi_1 = \eta_1$.

For a node v and a round t, we say that node v is *viable in round* t if $|\Gamma_{[t+1:t+3]}(v)| \le n^c$. (Here Γ is always defined with respect to H, instead of H_1.) Consider any given set B of nodes where $|B| < \min(n^c, N)$. Let $\tau = \tau_1\tau_2\ldots\tau_k$ be any sequence of nodes such that for all $1 \le s \le k$, node τ_s is in B and τ_s is viable in round $t_0 + 4s$. We will prove that for all such τ and under all given outcomes of R_1 and R_2, we have $\Pr[\eta = \tau] = \Pr[\phi = \tau]$ where the probability is defined over R_3. To prove this, it suffices to prove that $\Pr[\eta_{s+1} = \tau_{s+1} | \eta_s = \tau_s] = \Pr[\phi_{s+1} = \tau_{s+1} | \phi_s = \tau_s]$ for all $1 \le s \le k-1$. Condition on $\eta_s = \tau_s$ and $\phi_s = \tau_s$, we separately consider the following three cases for τ_{s+1}:

$\tau_{s+1} = \tau_s$. We reason about η_{s+1} first and ϕ_{s+1} next. By definition, τ_s is viable in round $t_0 + 4s$, so we have $|\Gamma_{[t_0+4s+1:t_0+4s+3]}(\tau_s)| \le n^c$. Thus in the round $(t_0 + 4s + 4)$, on node τ_s the condition at Line 17 in Algorithm 1 must be satisfied. (Recall that τ_s equals η_s, η_s is in ψ', and ψ' is defined when there are no sinks. Hence the second part of the condition at Line 17 is trivially satisfied.) Then we will have $\eta_{s+1} = \tau_{s+1}$ iff at Line 18, node τ_s either chooses itself or choose some node in $\Gamma_{[t_0+4s+1:t_0+4s+3]}(\tau_s) \setminus \Gamma_{[t_0+4s+1:t_0+4s+4]}(\tau_s)$. The probability of this happening is exactly $1 - \frac{|\Gamma_{[t_0+4s+1:t_0+4s+4]}(\tau_s)|}{n^c}$. Next we move on to ϕ_{s+1}. In round s of H_1, the set of neighbors of τ_s is exactly $\Gamma_{[t_0+4s+1:t_0+4s+4]}(\tau_s)$. Hence, the probability that $\phi_{s+1} = \tau_{s+1}$ is the probability that the $\frac{1}{n^c}$-BRW stays at node τ_s, which is exactly $1 - \frac{|\Gamma_{[t_0+4s+1:t_0+4s+4]}(\tau_s)|}{n^c}$. Hence $\Pr[\eta_{s+1} = \tau_{s+1} | \eta_s = \tau_s] = \Pr[\phi_{s+1} = \tau_{s+1} | \phi_s = \tau_s]$.

$\tau_{s+1} \ne \tau_s$ and $\tau_{s+1} \in \Gamma_{[t_0+4s+1:t_0+4s+4]}(\tau_s)$. We reason about η_{s+1} first and ϕ_{s+1} next. By same argument as above, in the round $(t_0 + 4s + 4)$, on node τ_s the condition at Line 17 in Algorithm 1 must be satisfied. Then we have $\eta_{s+1} = \tau_{s+1}$ iff at Line 18, node τ_s chooses τ_{s+1}. The probability of this happening is exactly $\frac{1}{n^c}$. On the other hand for ϕ_{s+1}, note that τ_s has $|\Gamma_{[t+4s+1:t+4s+4]}(\tau_s)| \le |\Gamma_{[t_0+4s+1:t_0+4s+3]}(\tau_s)| \le n^c$ neighbors in H_1. By definition of $\frac{1}{n^c}$-BRW, the probability of $\phi_{s+1} = \tau_{s+1}$ is also exactly $\frac{1}{n^c}$. Hence $\Pr[\eta_{s+1} = \tau_{s+1} | \eta_s = \tau_s] = \Pr[\phi_{s+1} = \tau_{s+1} | \phi_s = \tau_s]$.

$\tau_{s+1} \ne \tau_s$ and $\tau_{s+1} \notin \Gamma_{[t_0+4s+1:t_0+4s+4]}(\tau_s)$. Obviously $\Pr[\eta_{s+1} = \tau_{s+1} | \eta_s = \tau_s] = \Pr[\phi_{s+1} = \tau_{s+1} | \phi_s = \tau_s] = 0$.

We have proved that $\Pr[\eta = \tau] = \Pr[\phi = \tau]$. With the given set of nodes B, we can sum up the probabilities over all possibles τ, and we have $\sum_\tau \Pr[\phi = \tau] = \sum_\tau \Pr[\eta = \tau]$. On the other hand, since $|B| < \min(n^c, N)$, Theorem 4.2 tells us (via a trivial contradiction) that $\sum_\tau \Pr[\phi = \tau] \le \frac{1}{2}$. Hence we also have $\sum_\tau \Pr[\eta = \tau] \le \frac{1}{2}$. This means that for any given set of nodes B with $|B| < \min(n^c, N)$, and any given segment starting from node ψ'_{t_0+1}, with at most $\frac{1}{2}$ probability, both the following properties hold: i) the segment only visits nodes in B, and ii) for all $s \in [1, k]$, the node ψ'_{t_0+4s} is viable in the round $t_0 + 4s$.

We now move on to reason about all the segments in ψ'. For any segment starting from node ψ'_{t_0+1} for some t_0, we say that the segment is *nice* if at least one of the following three events happens: i) the segments before the current segment have already visited at least $\min(n^c, N)$ distinct nodes; ii) there is some $s \in [1, k]$ such that the node ψ'_{t_0+4s} is not viable in the round $t_0 + 4s$; or iii) the current segment visits at least one new node not visited by any of the previous segments. By letting B be the set of nodes visited by all the previous segments, our above reasoning implies that each segment is nice with at least $\frac{1}{2}$ probability (under all given outcomes of R_1 and R_2, and with the probability defined over R_3).

Although the events of different segments being nice are correlated, the conditional probability of each of the h segments being nice is always at least $\frac{1}{2}$, regardless of whether previous segments are nice. This allows us to apply a Chernoff bound despite the correlation, which shows that $\Pr[\text{less than } (\frac{1}{2} \cdot \frac{h}{2}) \text{ segments are nice}] \le \exp(-\frac{1}{8} \cdot \frac{h}{2}) < \frac{\epsilon}{8n}$, with the probability being defined over R_3. Hence, the sequence ψ' contains at least $\frac{h}{4}$ nice segments with probability (over R_3) at least $1 - \frac{\epsilon}{8n}$.

Connecting ψ to ψ'. We next draw a connection between ψ and ψ', to show that ψ visits some sink. Our following discussion will condition upon ψ' containing at least $\frac{h}{4}$ nice segments — or equivalently, we consider given outcome of R_3 under which ψ' contains at least $\frac{h}{4}$ nice segments. Note that given the outcome of R_3, the sequence ψ' is fixed. We separately consider two cases:

The sequence ψ' contains at least $\min(N, n^c)$ distinct nodes Recall that each node independently becomes a sink with probability $q = \min(1, \frac{8}{n^c}(\ln \frac{1}{\epsilon} + \ln(64n^2) + \ln\ln \frac{16n}{\epsilon}))$. The probability that there is no sink among these $\min(N, n^c)$ distinct nodes is at most $(1-q)^{\min(N,n^c)} < \frac{\epsilon}{8n}$, with the probability being taken over R_1. We will now only consider those outcomes of R_1 such that there is some sink among those nodes. Under such given outcomes of R_1 and R_3, if $\psi = \psi'$, then obviously ψ must visit a sink. If $\psi \neq \psi'$, then by definition of ψ and ψ', for ψ to be different from ψ', the sequence ψ must visit a sink.

The sequence ψ' contains less than $\min(N, n^c)$ distinct nodes. Recall that ψ' contains at least $\frac{h}{4}$ nice segments. We claim that there are at least $x = 2\sqrt{n^{1-c}}\ln(\frac{16n}{\epsilon})$ segments where each of them contains some node ψ'_t ($t \in [4, 4((h-1)(k+1)+k)]$) such that ψ'_t is non-viable in round t, and t is a multiple of 4. We prove this claim via a simple contradiction: If not, then by the definition of a nice segment, there will be at least $\frac{h}{4} - 2\sqrt{n^{1-c}}\ln(\frac{16n}{\epsilon}) = 2\sqrt{n^{1-c}}\ln(\frac{16n}{\epsilon}) > n^c$ segments that visits at least one new node not visited by previous segments in ψ'. This would then contradiction with ψ' containing less than $\min(N, n^c)$ distinct nodes.

Recall from above that under the given R_3, the only scenario where ψ can be different from ψ' is that ψ has visited a sink. The next will prove that the probability of $\psi = \psi'$ is at most $\frac{\epsilon}{8n}$, with the probability taken over R_1 and R_2. This implies that the probability of ψ visiting a sink is at least $1 - \frac{\epsilon}{8n}$.

To reason about the probability of $\psi = \psi'$, consider any given segment among the above x segments, and the corresponding node ψ'_t where ψ'_t is non-viable in round t. This means that $|\Gamma_{[t+1:t+3]}(\psi'_t)| > n^c$. Since t is a multiple of 4, Line 3 is executed in round $t+1$ on node ψ_t. Hence, by Lemma C.1, with probability at least $1 - \frac{\epsilon}{32n^2\ln(\frac{16n}{\epsilon})}$ over R_1, we have either $\frac{|S_{[t+1:t+2]}(\psi_t)|}{|S_{[t+1:t+1]}(\psi_t)|} \geq \frac{1}{2\sqrt{N^{1-c}}}$ or $\frac{|S_{[t+1:t+3]}(\psi_t)|}{|S_{[t+1:t+2]}(\psi_t)|} \geq \frac{1}{2\sqrt{N^{1-c}}}$. With a union bound, the earlier property holds for all x segments with probability at least $1 - \frac{\epsilon}{16n}$. Conditioned upon such an event, in order for ψ to remain the same as ψ' in a given segment among these x segments, the node ψ_t needs to fail in its attempt of directing the tuple to a neighboring sink at Line 4 to 15 in Algorithm 1. This happens with probability $(1 - \frac{1}{2\sqrt{N^{1-c}}})$ over R_2. Finally, taking all x such segments into account, the probability (over R_2) of ψ remaining the same as ψ' in all segments is at most $(1 - \frac{1}{2\sqrt{N^{1-c}}})^x = (1 - \frac{1}{2\sqrt{N^{1-c}}})^{2\sqrt{n^{1-c}}\ln(\frac{16n}{\epsilon})} < \frac{\epsilon}{16n}$.

Finally, a union bound across R_1, R_2, and R_3 shows that with probability at least $1 - \frac{\epsilon}{8n} - \max(\frac{\epsilon}{8n}, \frac{\epsilon}{8n}) = 1 - \frac{\epsilon}{4n} \geq 1 - \frac{\epsilon}{4N}$, the sequence ψ visits a sink. □

D PROOF FOR THEOREM 4.5

We first prove Lemma D.1 and D.2, and then use them to prove Theorem 4.5. Recall from Section 4.2 the notion of a *trial* in Algorithm 2, which is from Line 33 through 47 in Algorithm 2, and corresponds to one specific (n, d) pair.

Lemma D.1. *For any given $T \geq 4$, $N \geq 1$, and $\epsilon = \Omega(\frac{1}{\text{poly}(N)})$, with probability at least $1 - \frac{\epsilon}{2}$, Algorithm 2 outputs within $O(DN^{\frac{6}{7}}\text{polylog}(N))$ rounds on all nodes.*

Proof. Recall that Algorithm 2 defined the function $f(d, n) = 100(c'+1)dn^{\frac{6}{7}}\log^2(4n)\log\frac{4}{\epsilon}$. One can verify that this is an upper bound on the number of rounds needed by a trial for one (n, d) pair, namely, from Line 33 to 47 in Algorithm 2. Since $55f(D, 2N)\log^2 f(D, 2N) = O(DN^{\frac{6}{7}}\text{polylog}(N))\log^2\frac{4}{\epsilon} = O(DN^{\frac{6}{7}}\text{polylog}(N))$, it suffices to prove that all nodes output within $55f(D, 2N)\log^2 f(D, 2N)$ rounds. Let event E_1 be such that no nodes output within $54f(D, 2N)\log^2 f(D, 2N)$ rounds. We will prove that $\Pr[E_1] \leq \frac{\epsilon}{2}$. Proving this suffices since if at least one node outputs within $54f(D, 2N)\log^2 f(D, 2N)$ rounds, then by the mechanism at Line 50 in Algorithm 2 which floods the output, all nodes will output within $54f(D, 2N)\log^2 f(D, 2N) + D < 55f(D, 2N)\log^2 f(D, 2N)$ rounds.

Let β be the first trial such that throughout all rounds in that trial, every node u have $n \in [N, 2N]$, $d \geq D$, and $a_u = A$. We will first show that β must end by the end of round $54f(D, 2N)\log^2 f(D, 2N)$, and then show that at least one node outputs by the end of the trial β with probability at least $1 - \frac{\epsilon}{2}$. This will then complete the proof.

Define Line 31 through 48 in Algorithm 2 to be an *iteration*, where the i-th iteration has a "budget" of 2^i rounds for each trial within that iteration. It is easy to verify that the i-th iteration ($i \geq 1$) takes at most $1.5i2^i$ rounds. Furthermore, we claim that the i-th iteration for $i \geq \log f(D, 1)$ must take at least D rounds: Since $f(D, 1) \leq 2^i$ when $i \geq \log f(D, 1)$, the first trial in that iteration will use $d \geq D$ and $n = 1$, and that trial will already take $f(d, 1) \geq d \geq D$ rounds. This then implies that in the i-th iteration where $i \geq \log f(D, 1) + 1$, all node u must have $a_u = A$. Next, let $i_0 = \lceil\max(\log f(D, 1) + 1, \log f(D, 2N))\rceil$. Since $i_0 \geq \log f(D, 1) + 1$ and since $f(D, 2N) \leq 2^{i_0}$, the i_0-th iteration must contain some trial during which all node u use $n \in [N, 2N]$, $d \geq D$, and $a_u = A$. The trial β can be either this trial, or some earlier trial. In either case, the trial β must finish by the end of the i_0-th iteration, which is no later than round $\sum_{i=1}^{i_0} 1.5i2^i \leq i_0 \cdot 1.5 \cdot i_0 \cdot 2^{i_0} \leq 1.5 \cdot (2 + \log f(D, 2N))^2 \cdot 2^{2+\log f(D,2N)} \leq 54f(D, 2N)\log^2 f(D, 2N)$.

We have proved that β must finish by the end of round $54f(D, 2N)\log^2 f(D, 2N)$. Recall that E_1 is the event that no nodes output within $54f(D, 2N)\log^2 f(D, 2N)$ rounds. We next prove $\Pr[E_1] \leq \frac{\epsilon}{2}$. Let E_2 denote the event that no node outputs before trial β. Then we have $\Pr[E_1] \leq \Pr[\text{no node outputs during } \beta \mid E_2] \times \Pr[E_2] \leq \Pr[\text{no node outputs during } \beta \mid E_2]$. The following proves that $\Pr[\text{no node outputs during } \beta \mid E_2] \leq \frac{\epsilon}{2}$, by showing that $\Pr[\text{some node outputs during } \beta \mid E_2] \geq 1 - \frac{\epsilon}{2}$. All our following discussion will condition upon E_2.

As no node has output before trial β, all nodes must invoke Algorithm 1 during the trial β. By Lemma 4.4 and with a union bound across all N nodes, with probability at least $1 - \frac{\epsilon}{4}$, every tuple will reach some sink. Since each node selects itself as a sink with probability $q = \min(1, \frac{8}{n^c}(\ln\frac{1}{\epsilon} + \ln(64n^2) + \ln\ln\frac{16n}{\epsilon}))$, by Chernoff bound, with probability at least $1 - \exp(-\frac{1}{3}qN) \leq \frac{\epsilon}{4}$, there are at most $2qN$ sinks. Then, a union bound gives that with probability at least $1 - \frac{\epsilon}{2}$, every tuple reaches some sink and there are at most $2qN$ sinks. It suffices now to show that conditioned on every tuple reaches some sink, and conditioned on there are at most $2qN$ sinks, for every node w, the condition at Line 46 of the trial β must be satisfied. This will then cause node w to output by the end of the trial β. The following proves this.

First, at Line 46 of the trial β, we must have $\texttt{flag}_w$ = False. This is because in trial β, all node u choose p_u to be the smallest prime number such that p_u is at least $8j^2a_u^2\frac{1}{\epsilon} = 8j^2A^2\frac{1}{\epsilon}$, implying that p_u is the same for all u. This means that no node will set its flag due to seeing conflicting p values. Additionally, since $p_u > A \geq N$, the condition at Line 39 will not be satisfied on any node u, and hence the flag will not be set at that line either.

Second, we claim for every node w at Line 46 of the trial β, we have $\texttt{allTuple}_w = \sum_u \texttt{tuple}_u$ with the sum taken over all nodes in the system: Since $d \geq D$ on all nodes during the trial β, one can verify that by the end of each loop from Line 59 to 62 in Algorithm 3, every node will have selected the same id for the node with that id to flood its aggregate tuple. Then the selected node will flood its tuple for d rounds from Line 72 to 75. Since $d \geq D$, the flooding will be received by all nodes, and this selected node will not join future selections. As there are at most $2qN$ sinks, and since Algorithm 3 selects $\lceil 2qn \rceil \geq 2qN$ nodes to flood, every sink will get a chance to flood its aggregated tuple. Together with the fact that every tuple reaches some sink, we have $\texttt{allTuple}_w = \sum_u \texttt{tuple}_u$ with the sum taken over all nodes in the system.

Finally, since $\texttt{allTuple}_w = \sum_u \texttt{tuple}_u$, the value $\texttt{allTuple}_w.\texttt{count}$ must be positive, and we further have $\texttt{allTuple}_w.\texttt{rand\_num} = \sum_u \texttt{rand\_num}_u = 0 \mod p_w$. This then enables w to satisfy the condition Line 46 in the trial β. □

The following Lemma D.2 upper bounds the error in our soundness checking mechanism. The proof follows from the intuition in Section 3.4. Note that the lemma holds regardless of whether some nodes (other than w) have already output before the given trial:

Lemma D.2. *Consider any $T \geq 1$. For any given trial in Algorithm 2, the probability that a given node w outputs a wrong result at Line 46 of Algorithm 2 is at most $\frac{1}{p_w}$.*

Proof. In this proof, all the line numbers refer to either those lines executed in the given trial in Algorithm 2, or the lines executed in the corresponding invocations of Algorithm 1 and 3 during that trial. We say that a node is *inactive* in the trial if either it has output prior to this trial, or it outputs at Line 40 of this trial. Otherwise, the node is *active* in the trial. If w is inactive in the trial, then the lemma trivially holds. Hence we only need to prove for the case where w is active. Fix the randomness used in Algorithm 1. We define the set S_1 to contain all the nodes from which node w has collected their respective tuples by Line 46. Specifically, consider any node u and the corresponding path ψ of node u's tuple.[6] Then node u is in S_1 iff in the trial, node w has received the aggregate tuple from $\psi_\perp$ — or more precisely, during this trial w has reached Line 76 in Algorithm 3 (once) with $\texttt{max\_sink\_id}_w = \psi_\perp$ and $\texttt{flooding}_w \neq \texttt{null}$. All nodes u in S_1, and their corresponding nodes $\psi_\perp$, must be active, since otherwise node w would not have collected node u's tuple.

By the definition of S_1, one can verify from the pseudo-code that immediately after Line 44 of Algorithm 2, we have $\texttt{allTuple}_w = \sum_{v \in S_1} \texttt{tuple}_v$, where the summation of multiple tuples is obtained by summing the $\texttt{count}$ and $\texttt{rand\_num}$ fields separately. Node w outputs a wrong result in this trial only if all the following holds at Line 46: i) $S_1 \neq \emptyset$ and $S_1 \neq V$ where V is the set of all N nodes, ii) $\texttt{flag}_w$ = False, and iii) $\texttt{allTuple}_w.\texttt{rand\_num} = 0 \mod p_w$. To prove the lemma, it suffices to show that $\Pr[\texttt{allTuple}_w.\texttt{rand\_num} = 0 \mod p_w \mid (S_1 \neq V \text{ and } S_1 \neq \emptyset \text{ and } \texttt{flag}_w = \text{False})] = \frac{1}{p_w}$. Our following discussion will always condition upon the event that $S_1 \neq V$ and $S_1 \neq \emptyset$ and $\texttt{flag}_w$ = False.

We first show that for all node $u \in S_1$, it holds that $p_u = p_w$ and $\texttt{flag}_u$ = False immediately before Line 42 of Algorithm 2. Consider any $u \in S_1$, and let ψ be the corresponding path traversed by the tuple from u. By the definition of S_1, node w must have received the tuple from node $\psi_\perp$ at Line 44 in Algorithm 2. Since $\texttt{flag}_w$ = False immediately after Line 44 of Algorithm 2, we must have $p_{\psi_\perp} = p_w$ and $\texttt{flag}_{\psi_\perp}$ = False immediately before Line 44. In turn, since $\texttt{flag}_{\psi_\perp}$ = False immediately before Line 44, we must have $p_u = p_{\psi_\perp} = p_w$ and $\texttt{flag}_u$ = False immediately before Line 42.

Next, the topology of the network in the first round of this trial must be connected. Since $S_1 \neq \emptyset$ and $S_1 \neq V$, there must exist some node $v_0 \in S_1$ such that in the first round of this trial, node v_0 has at least one neighbor that is not in S_1. Let the non-empty set S' be the set of neighbors of node v_0 that are not in S_1. We earlier showed that $\texttt{flag}_{v_0}$ = True immediately before Line 42, which implies that $|S'| < p_{v_0} = p_w$, by Line 39 of Algorithm 2. Hence $1 \leq |S'| \leq p_w - 1$.

Fix all the randomness used by all nodes, except for the randomness used by v_0 for generating its random share x_{v_0}. Then at Line 46 of Algorithm 2 on node w, the value of $\texttt{allTuple}_w.\texttt{rand\_num}$ will be a function of x_{v_0}. Putting it another way, we must have $\texttt{allTuple}_w.\texttt{rand\_num} = g(x_{v_0})$ for some function $g()$, and $g(0)$ is the value of $\texttt{allTuple}_w.\texttt{rand\_num}$ when for each neighbor, node v_0 sends 0 as a share to that neighbor and keeps a share of p_w for itself. Hence by the reasoning in Section 3.3, we have $g(x_{v_0}) = |S'|(p_w - x_{v_0}) + g(0) \mod p_w$. We now have $\Pr[\texttt{allTuple}_w.\texttt{rand\_num} = 0 \mod p_w] = \Pr[|S'|(p_w - x_{v_0}) + g(0) = 0 \mod p_w] = \Pr[|S'|x_{v_0} = g(0) \mod p_w]$. Since p_w is a prime number, $1 \leq |S'| \leq p_w - 1$, and x_{v_0} is uniformly random in $[0, p_w - 1]$, we have $\Pr[|S'|x_{v_0} = g(0) \mod p_w] = \frac{1}{p_w}$. □

We restate and prove Theorem 4.5 below:

Theorem 4.5. *For all given $T \geq 4$, $N \geq 1$, and $\epsilon = \Omega(\frac{1}{\text{poly}(N)})$, with probability at least $1 - \epsilon$, Algorithm 2 outputs N on all nodes within $O(DN^{\frac{6}{7}}\text{polylog}(N))$ rounds.*

Proof. By Lemma D.2, for each trial and each node u, the probability that u outputs wrongly at Line 46 in that trial is at most $\frac{1}{p_u}$. In a trial, node u sets p_u to be the smallest prime number such that $p_u \geq \frac{4a_u^2}{\epsilon} \times (2j^2)$, where $a_u \geq u$ is the largest id that node u has seen and j is the number of trials u has done. Hence, across all trials (even if there are infinite number of trials), the probability of u outputting wrongly at Line 46 is at most $\sum_{j=1}^{\infty} \frac{\epsilon}{4a_u^2 \times 2j^2} \leq \frac{\epsilon}{4a_u^2}$. A union bound across all nodes then shows that the probability of some node outputting a wrong result at Line 46 is at most $\sum_{u=1}^{N} \leq \frac{\epsilon}{4u^2} < \frac{\epsilon}{2}$. Finally, it is easy to verify from the pseudo-code that if no node outputs a wrong result at Line 46, then no node can possibly output any wrong result at Line 40 either. Hence with probability at

[6]Notice that the notion of a path is still well-defined even when some nodes are inactive. In particular, by the definition of a path, if the path reaches a node that did not invoke Algorithm 1, then the path remains at that node until the end.

least $1-\frac{\epsilon}{2}$, Algorithm 2 never outputs a wrong result on any node. Taking a union bound with the probability in Lemma D.1 then shows that with probability at least $1-\epsilon$, Algorithm 2 outputs N on all nodes within $O(DN^{\frac{6}{7}}\text{polylog}(N))$ rounds. Finally, in all those rounds, one can verify that message size in the algorithm is always $O(\log N)$. □

E PROOF FOR THEOREM 5.1

We restate and prove Theorem 5.1 below:

Theorem 5.1. *For all $T \geq 4$, $N \geq 1$, and $\epsilon = \Omega(\frac{1}{\text{poly}(N)})$, with probability at least $1-\epsilon$, the algorithm described in Section 5 outputs N and also terminates on all nodes, within $O(DN^{\frac{6}{7}}\text{polylog}(N))$ rounds.*

Proof. We first do some preparation, then prove that the algorithm must output N correctly, and finally show that it terminates within $O(DN^{\frac{6}{7}}\text{polylog}(N))$ rounds.

Preparation. The proofs of Lemma D.1 and Theorem 4.5 show that there exists constants $a_1 \geq 1$ and $a_2 \geq 1$ such that for all $N \geq 1$ and $\epsilon = \Omega(\frac{1}{\text{poly}(N)})$, with probability at least $1-\epsilon$, Algorithm 2 outputs the correct result on all nodes within $(a_1 DN^{\frac{6}{7}})(\log^{a_2} N)(\log^2 \frac{4}{\epsilon})$ rounds. For given D, N, and ϵ, define $i_0 \geq 1$ to be the smallest integer i such that

$$2^i \quad \geq \quad (a_1 DN^{\frac{6}{7}})(\log^{a_2} N)(\log^2 \frac{4}{\frac{\epsilon}{2^{i+1}}})$$

It is easy to verify that i) such i_0 must exist, ii) the above inequality holds for all $i \geq i_0$, and iii) $2^{i_0} = O(DN^{\frac{6}{7}}\text{polylog}(N)\text{polylog}(\frac{1}{\epsilon}))$. For $\epsilon = \Omega(\frac{1}{\text{poly}(N)})$, we further have $2^{i_0} = O(DN^{\frac{6}{7}}\text{polylog}(N))$. We will only be concerned with the invocations of Algorithm 2 for computing M in epoch 1 through $i_0 + 2 + \log N$. Recall that we allow epoch i ($i \geq 1$) to have an error probability of $\epsilon'_i = \frac{\epsilon}{2^{i+1}}$. For $1 \leq i \leq i_0 + 2 + \log N$ and for $\epsilon = \Omega(\frac{1}{\text{poly}(N)})$, we always have $\epsilon'_i \geq \frac{\epsilon}{2^{i_0+3+\log N}} = \Omega(\frac{1}{\text{poly}(N)})$.

Now we can invoke Theorem 4.5 for all the invocations of Algorithm 2 for computing M in epoch 1 through $i_0 + 2 + \log N$, and for the single invocation of Algorithm 2 for computing N with an error probability of $\epsilon'_0 = \frac{\epsilon}{2}$. A union bound immediately shows that with probability at least $1 - \sum_{i=0}^{i_0+2+\log N} \epsilon'_i \geq 1-\epsilon$, every such invocation must return the correct result on all nodes within $(a_1 DN^{\frac{6}{7}})(\log^{a_2} N)(\log^2 \frac{4}{\epsilon'_i})$ rounds, if all nodes participate throughout that invocation. The remainder of our proof will condition upon this (good) event happening.

Outputting N correctly. We can now prove that the algorithm outputs the correct count N. Let round r_1 be the first round during which some node generates an output for count. Recall that whenever a node u outputs the count, it also floods a TERM1 message, and this TERM1 message is also viewed as being received by node u itself immediately in the current round. Hence round r_1 is also the first round during which some node receives a TERM1 message. Define round $r_2 \geq r_1$ to be the last round during which some node generates an output for count. Hence by round r_2, all nodes in the system have received at least one TERM1 message. Let round r_3 be the first round during which some node terminates. We claim that $r_3 \geq r_2$. To see why, consider any node u that terminates in round r_3. If u terminates according to TERM1, then the source of that TERM1 message must have initiated the flooding of the TERM1 message in round $r_3 - N$. Since no node terminates before round r_3, this flooding must have reached all nodes by round r_3. A node must generate an output for count upon receiving this TERM1 messages, if it has not yet generated an output. Hence $r_2 \leq r_3$.

Next, if u terminates according to TERM2, then we claim that r_3 must be before or in epoch $i_0 + \log N$. We prove this claim via a simple contradiction, and assume r_3 is after epoch $i_0 + \log N$. This means no node terminates in or before epoch $i_0 + \log N$. Then by the definition of i_0 and by the time complexity of Algorithm 2, the single invocation of Algorithm 2 for computing N must have generated the correct result on all nodes by the end of epoch i_0. Hence every node must receive some TERM1 message (potentially from itself), which instructs it to terminate in some round that is no later than N rounds after the end of epoch i_0. That round must be in or before epoch $i_0 + \log N$. This would then contradict with u being the first node that terminates.

We have shown that r_3 must be before or in epoch $i_0 + \log N$. Our earlier discussion on i_0 tells us that the result generated by an invocation of Algorithm 2 prior to round r_3 must be correct, if all nodes have participated in that invocation. Since u terminates in round r_3 according to TERM2, the source of the corresponding TERM2 message must have previously invoked Algorithm 2 to compute M and observed that $M = N$. Since no node terminates before round r_3, all nodes must have participated in that invocation, and hence such computation must be correct. This implies that all nodes must have received a TERM1 message (and hence output the count N) by round r_3.

So far we have proved that $r_1 \leq r_2 \leq r_3$. This means that no node ever terminates, before all nodes have generated the output for count. Since the invocation of Algorithm 2 for computing N must generate a correct result for count if all nodes participate in that invocation, we know that all nodes must output N correctly.

Terminating within $O(DN^{\frac{6}{7}}\text{polylog}(N))$ rounds. Finally, we prove that the algorithm terminates on all nodes within $O(DN^{\frac{6}{7}}\text{polylog}(N))$ rounds. By the definition of i_0, the number of rounds in each epoch, and the time complexity of Algorithm 2, we know that in every epoch from epoch i_0 through $i_0 + 2$, the invocation of Algorithm 2 for computing M in that epoch must generate the correct result on all nodes by the end of that epoch, if all nodes have participated in that invocation.

Furthermore, we have already shown earlier that no node ever terminates, before all nodes have generated the output for count. Such output is either generated by the single invocation of Algorithm 2 for computing N, or generated upon receiving a TERM1 message. In either cases, the output (if generated) must be correct since no node has terminated when the output is generated. In addition, we claim that every node must have generated the output for count by the end of epoch i_0. This is because otherwise no node would have terminated in or before epoch i_0, which in turn implies that the single invocation of Algorithm 2 for computing N must generate output on all nodes by the end of epoch i_0.

Hence every node must generate the output for count and also receive some TERM1 message, by the end of epoch i_0. This TERM1 message will instruct the node to terminate in some round that is no later that N rounds after the end of epoch i_0. The total number of rounds, from the beginning of the execution until the end of epoch i_0, is at most $2 \cdot 2^{i_0}$. Regardless of whether a node receives any TERM2 message, the node hence must terminate by round $2 \cdot 2^{i_0} + N$. If $D \geq N^{\frac{1}{7}}$, then we immediately have $2 \cdot 2^{i_0} + N = O(DN^{\frac{6}{7}}\text{polylog}(N)) + N = O(DN^{\frac{6}{7}}\text{polylog}(N))$, and we are done.

We still need to prove for $D < N^{\frac{1}{7}}$. Let round r_4 be the first round during which a node terminates according to TERM1. Define $r_4 = \infty$ if there is no such node. If round r_4 is no later than the end of epoch $i_0 + 2$, then we must have $2 \cdot 2^{i_0+2} \geq r_4 \geq N$. We explained earlier that regardless of whether a node receives any TERM2 message, the node must terminate by round $2 \cdot 2^{i_0} + N$. Since $2 \cdot 2^{i_0+2} \geq N$, this means every node must terminate by round $2 \cdot 2^{i_0} + N \leq 2 \cdot 2^{i_0} + 2 \cdot 2^{i_0+2} = O(DN^{\frac{6}{7}}\text{polylog}(N))$, and we are done.

The only remain case is when $D < N^{\frac{1}{7}}$ and round r_4 is after epoch $i_0 + 2$. By the definition of r_4, in the first $i_0 + 2$ epochs, no node terminates according to TERM1. We claim that in the first $i_0 + 2$ epochs, there must exist some node that terminates according to TERM2.

We prove this claim via a simple contradiction, and assume that there is no such node. Since r_4 is after epoch $i_0 + 2$, this means that no node terminates in the first $i_0 + 2$ epochs. We explained earlier that every node must generate the correct output for count and also receive some TERM1 message, by the end of epoch i_0. In epoch $i_0 + 1$, the invocation of Algorithm 2 for computing M must generate the correct result of $M = N$ on all nodes, by the end of that epoch. Hence in epoch $i_0 + 1$, some node u will initiate the flooding of a TERM2 message. Node u itself must also receive this TERM2 message, which will cause it to terminate at most $N^{\frac{1}{7}}$ rounds after the end of epoch $i_0 + 1$. Since epoch $i_0 + 2$ has at least $N^{\frac{1}{7}}$ rounds, node u must terminate by the end of epoch $i_0 + 2$, which is a contradiction.

We have proved that in the first $i_0 + 2$ epochs, there must exist some node that terminates according to TERM2. Let r_5 be the first round during which some node terminates according to TERM2, and let u be one such node. The source of that TERM2 message must have initiated the flooding of that TERM2 message in round $r_5 - N^{\frac{1}{7}}$. Since no node terminates (either according to TERM1 or TERM2) before round r_5 and since $D < N^{\frac{1}{7}}$, this flooding must have reach all nodes by round r_5. In turn, all nodes will terminate in round r_5, which is no later than the end of epoch $i_0 + 2$. The total number of rounds before termination is at most $2 \cdot 2^{i_0+2} = O(DN^{\frac{6}{7}}\text{polylog}(N))$, and we are done. □