On the Power of Randomization in Distributed Algorithms in Dynamic Networks with Adaptive Adversaries*

Irvan Jahja
National University of Singapore
irvan@comp.nus.edu.sg

Haifeng Yu
National University of Singapore
haifeng@comp.nus.edu.sg

Ruomu Hou
National University of Singapore
houruomu@comp.nus.edu.sg

Abstract

This paper investigates the power of randomization in *general* distributed algorithms in *dynamic networks* where the network's topology may evolve over time, as determined by some *adaptive adversary*. In such a context, randomization may help algorithms to better deal with i) "bad" inputs to the algorithm, and ii) evolving topologies generated by "bad" adaptive adversaries. We prove that randomness offers limited power to better deal with "bad" adaptive adversary. We define a simple notion of *prophetic adversary* for determining the evolving topologies. Such an adversary accurately predicts all randomness in the algorithm beforehand, and hence the randomness will be useless against "bad" prophetic adversaries. Given a randomized algorithm P whose time complexity satisfies some mild conditions, we prove that P can always be converted to a new algorithm Q with comparable time complexity, even when Q runs against prophetic adversaries. This implies that the benefit of P using randomness for dealing with the adaptive adversaries is limited.

1 Introduction

Background. Understanding the power of randomization has long been a key goal in algorithms research. Over the years, researchers have obtained many interesting results on the power of randomization, such as in centralized algorithms (e.g., [25]), in parallel algorithms (e.g., [21]), and in algorithms in static networks (e.g., [7, 8, 10, 11, 22]). This paper aims to gain deeper insights into the power of randomization in *general* distributed algorithms in *dynamic networks* with *adaptive adversaries*. Dynamic networks [4, 6, 19, 23] model communication networks whose topologies may change over time, and has been a growing research topic in distributed computing. While randomization has been used extensively to solve various specific problems in dynamic networks (e.g., [17, 19]), prior works have not focused on the power of randomization in *general* distributed algorithms in dynamic networks (i.e., to what extent randomized algorithms can outperform deterministic ones).

Our setting. We consider a synchronous dynamic network with a fixed set of n nodes. The network topology in each round is some arbitrary connected and undirected graph as determined by an *adaptive adversary*,

*The first two authors of this paper are alphabetically ordered. This is the technical report version of [13]. The final authenticated version is available online at https://doi.org/10.1007/978-3-030-57675-2_24.

and we adopt the following commonly-used model [16, 18, 26]: The adaptive adversary decides the round-r topology based on the algorithm's coin flip outcomes so far (i.e., up to and including round r). The adaptive adversary does not see the coin flip outcomes in round $r+1$ or later. We follow the communication model in [14, 26]: In each round, a node may choose to either send an $O(\log n)$ size message (i.e., the broadcast $\mathcal{CONGEST}$ model [24]) or to receive. A message sent is received, by the end of that round, by all the receiving neighbors of the sender in that round. Each node has some *input* of arbitrary size and a unique id between 0 and $n-1$. We consider *general distributed computing problems* modeled as some arbitrary function of the n input values (as an input vector). The output of the function is also a vector of length n, and node i should output the $(i+1)$-th entry of that vector. There is no constraint on the output size. An algorithm in this paper always refers to some algorithm for solving some distributed computing problem as modeled in the above way. Note that many problems that are not typically defined as functions, such as computing a unique minimum spanning tree (of some input graph) and token dissemination [9, 17], can nevertheless be modeled as a function. The *time complexity* is defined to be the number of rounds needed for all nodes to output. An algorithm P's *time complexity*, denoted as $\text{tc}_P(n, d)$, corresponds to its time complexity under the *worst-case scenario*. Here the *worst-case scenario* consists of i) the *worst-case input (vector)*, and ii) the *worst-case adaptive adversary* for generating dynamic networks with at most n nodes and at most d dynamic diameter. The dynamic diameter [18] of a dynamic network, intuitively, is the minimum number of rounds needed for a node u to causally influence another node v, when considering the worst-case u and v in the dynamic network. Section 2 gives a full description of the model.

Randomness in dynamic networks. For any given deterministic algorithm, informally, let us call its corresponding worst-case scenario as a "bad" scenario. A "bad" scenario for one deterministic algorithm may very well not be a "bad" scenario for other deterministic algorithms. Since a randomized algorithm is a distribution of deterministic algorithms, intuitively, randomization potentially helps to better deal with all those "bad" scenarios. For algorithms in dynamic networks, a "bad" scenario consists of a "bad" input and a "bad" adaptive adversary.

For dealing with "bad" inputs in dynamic networks, it is not hard to see that randomization can help to reduce the time complexity *exponentially*. For example, consider the two-party communication complexity (CC) problem EQUALITY [20]. Let m be the size of the input, then EQUALITY has a randomized CC of $O(\log m)$ bits, and a deterministic CC of $\Omega(m)$ bits [20]. Under our setting of dynamic networks with congestion, this exponential gap in the CC of EQUALITY directly translates to an exponential gap in the time complexity.

A quick conjecture? For dealing with "bad" adaptive adversaries, on the other hand, one may quickly conjecture that randomness has limited power: On the surface, since the randomness in round r is already visible to the adaptive adversary when it chooses the round-r topology, such randomness offers no help for better dealing with the round-r topology. But a deeper look shows that the randomness in round r could potentially help the algorithm to better deal with round-r' ($1 \leq r' < r$) topologies: Consider an example algorithm that uses the first $r-1$ rounds to flood a certain token in the network, where $r-1$ can be smaller than the network's dynamic diameter d. Let S be the set of nodes that have received the token by the end of round $r-1$. In round r and later, the algorithm may want to estimate the size of S (e.g., to estimate whether the token has reached some constant fraction of the nodes). The adaptive adversary can influence S, by manipulating the topologies in the first $r-1$ rounds. But by the end of round $r-1$, the set S will be fixed — effectively, the adaptive adversary has now committed to S. The algorithm's randomness in round r and later is independent of S. Thus for the remainder of the algorithm's execution (i.e., the part starting from round r), S can be viewed as a "midway input". The randomness in the remainder of the algorithm's execution can potentially help to better deal with such "midway inputs", and hence help indirectly to better

deal with the adaptive adversary's "bad" behavior in the first $r-1$ rounds.

Given such possibility, it is unclear whether the earlier quick conjecture holds or whether it may even be wrong. Resolving this will be our goal.

Our results for LV algorithms. As our main novel result, we prove that the earlier conjecture does hold, subject to some mild conditions on the algorithm's time complexity. (We will fully specify these mild conditions later.) As one will see later, proving this conjecture is far from trivial. We first need to expose the power of randomization for dealing with adaptive adversaries, and in particular, to properly isolate such power from the power of randomization for dealing with inputs. It is not immediately obvious how to do this since the same randomness may be used for dealing with both inputs and adaptive adversaries. To this end, we define a simple notion of *prophetic adversary* for determining the dynamic network. A *prophetic adversary* first sees (accurately predicts) all coin flip outcomes of a randomized algorithm in all rounds, and then decides the dynamic network (i.e., topologies in each round). This enables a prophetic adversary to always choose the worst-case dynamic network for the given coin flip outcomes. Hence the randomness in the algorithm can never help to better deal with dynamic networks generated by "bad" prophetic adversaries.

Now let us consider adaptive adversaries that generate dynamic networks with at most n nodes and at most d dynamic diameter. Let P be any Las Vegas (LV) algorithm whose time complexity (under the worst-case among all such adaptive adversaries) is $\text{tc}_P(n,d) = \Theta(f(n)\cdot g(d))$, for some $f(n)$ and $g(d)$ where there exists some constant a such that $\Omega(1) \leq f(n) \leq O(n^a)$ and $\Omega(d) \leq g(d) \leq O(d^a)$.[1,2] We prove (Theorem 1 and 2) that P can always be converted into another LV algorithm Q whose time complexity under worst-case *prophetic adversaries* is $O(\text{polylog}(n))\cdot\text{tc}_P(n,d)$. This means that even when the adversary accurately predicts all randomness in Q, Q's time complexity is still only $O(\text{polylog}(n))\cdot\text{tc}_P(n,d)$. In turn, the benefit of randomization (in P) for dealing with the adaptive adversaries is at most to reduce the complexity by a $O(\text{polylog}(n))$ factor. This proves the earlier conjecture affirmatively (under the previous mild conditions).

The more general version (Theorem 2) of our results actually hold for P as long as P's time complexity is upper bounded by some polynomial — namely, as long as there exists some constant a such that $\Omega(1) \leq \text{tc}_P(n,n) \leq O(n^a)$, and without any other constraints on $\text{tc}_P(n,d)$. Here for any given LV algorithm P, we can construct another LV algorithm Q whose time complexity under prophetic adversaries is $O((d\log^3 n)\times \text{tc}_P(n, a'\log n))$ for some constant a'. Hence in this more general case, our results imply that the power of randomization (in P) for dealing with adaptive adversaries is at most a $O(d\log^3 n)$ multiplicative factor when $d \geq a'\log n$.

Finally, the above shows that for dealing with adaptive adversaries, the power of randomization is inherently limited. This suggests that if an algorithm is not using randomness for better dealing with the inputs, we should be able to derandomize it efficiently. We show how this can be done for LV algorithms in Section 4.2.

Our results for MC algorithms. We have also obtained similar results for Monte Carlo (MC) algorithms. We defer the details to Appendix G, and provide a summary here. Consider any constant $\epsilon \in (0, 1-\delta)$ and any δ-error Monte Carlo (MC) algorithm P such that $\text{tc}_P(n,d) = \Theta(f(n)\cdot g(d))$, where there exists some constant a such that $\Omega(n) \leq f(n) \leq O(n^a)$ and $\Omega(d) \leq g(d) \leq O(d^a)$. Then we can always construct another $(\delta+\epsilon)$-error MC algorithm Q for solving the same problem and whose time complexity under worst-case prophetic adversaries is $O(\text{polylog}(n))\cdot\text{tc}_P(n,d)$. A more general version of this result holds for P as long as there exists some constant a such that $\Omega(1) \leq \text{tc}_P(n,n) \leq O(n^a)$, and without any

[1] Throughout this paper, $\Omega(h_1(x)) \leq h_2(x) \leq O(h_3(x))$ means $h_2(x) = \Omega(h_1(x))$ and $h_2(x) = O(h_3(x))$.

[2] Some quick examples of $\text{tc}_P(n,d)$ satisfying such a condition include $\Theta(d\log n)$, $\Theta(dn\log d)$, and $\Theta(d^{1.1}n^{1.5})$. On the other hand, $\text{tc}_P(n,d) = \Theta(d^2+\sqrt{n})$ does not satisfy the condition.

other constraints on $\mathrm{tc}_P(n, d)$. In this more general version, our algorithm Q will have a time complexity of $O((d \log^3 n) \times \mathrm{tc}_P(n, a' \log n) + n \log^2 n)$.

Our techniques. To obtain Q from P, we essentially need to "derandomize" the part of P's randomness used to deal with the adaptive adversaries. It turns out that such randomness is less amenable to typical derandomization methods such as pairwise independence, conditional expectation, or network decomposition. This motivated us to take a rather different route from prior derandomization efforts [5, 7, 8, 10, 11, 21, 22, 25].

Specifically, we will have Q simulate the execution of P against some adaptive adversary $\alpha^{\epsilon,t}$ that we construct. (Namely, Q simulates both P and $\alpha^{\epsilon,t}$.) To work against prophetic adversaries, Q will perform the simulation by only doing simple floodings. We will carefully design $\alpha^{\epsilon,t}$ so that: i) Q can efficiently simulate the execution of P against $\alpha^{\epsilon,t}$ via simple floodings, ii) the dynamic networks generated by $\alpha^{\epsilon,t}$ will have $O(\log n)$ dynamic diameter with at least $1 - \epsilon$ probability, and iii) there are some sufficient conditions, which can be efficiently checked in a distributed fashion, for guaranteeing the $O(\log n)$ dynamic diameter. Next, a central difficulty in the simulation is that Q does not know the dynamic diameter of the dynamic network over which it runs, which will cause various problems in the floodings. While Q can naturally use the standard doubling-trick to guess the dynamic diameter, the challenge is that Q cannot easily tell whether the guess is correct. As a result, we will carefully reason about the properties of the simulation when the guess is wrong, and design Q correspondingly.

Other types of adversaries. The adaptive adversaries in this paper (also called *strongly adaptive adversaries* [2, 9, 16, 18]) are not the only type of adversaries in dynamic networks. A more general notion is *z-oblivious adversaries* [1], which can see all randomness up to and including round $r - z$ when deciding the round-r topology. Prophetic adversaries and strongly adaptive adversaries correspond to z-oblivious adversaries with $z = -\infty$ and $z = 0$, respectively. Researchers have also considered 1-oblivious adversaries (also called *weakly adaptive adversaries* [9, 12]) and ∞-oblivious adversaries (also called *oblivious adversaries* [2, 3, 12]). The results in this paper are only for 0-oblivious adversaries, but our proofs are already non-trivial. The power of the algorithm's randomization will likely increase as z increases. On the other hand, we suspect that our conjecture could potentially be extended to 1-oblivious adversaries — we leave this to future work.

Related Work. Randomization has been extensively used for solving various specific problems in dynamic networks (e.g., [17, 19]). However, prior works have not focused on the power of randomization in *general* distributed algorithms in dynamic networks. On the other hand, there have been many works on the power of randomization in other settings, and we discuss the most relevant ones in the following.

In centralized setting, an *online algorithm* processes a sequence of *requests*, one by one. In this context, an *adaptive adversary* generates the i-th request after seeing the algorithm's behavior (and coin flips) on request 1 through $i - 1$. It is well-known [5] that randomized online algorithms against adaptive adversaries can always be effectively derandomized. However, the measure of goodness for online algorithms is competitive ratio, and hence the derandomized algorithm can afford to have exponential computational complexity. In our distributed setting, adopting the techniques from [5] would require us to collect all the n input values to one node, which would result in unbounded time complexity (i.e., number of rounds) since the sizes of our inputs are not constrained. Due to these fundamental differences, our results and techniques are all quite different from [5].

More recently, there have been a series of breakthrough results on derandomizing distributed algorithms in *static networks* [7, 8, 10, 11, 15, 22]. These derandomization results are all for *local algorithms*, where the output (or the correctness of the output) of a node only depends on its small neighborhood instead of on the entire network. In fact, many of them consider algorithms with $O(1)$ time complexity. Such a notion of local

n	number of nodes
d	(dynamic) diameter
$\alpha^{\epsilon,t}$, $\beta^{\epsilon,t}$	specific adaptive adversaries – defined in Section 3 and Appendix D.1
γ^G	specific adaptive adversary that always generates the given dynamic network G
τ	(general) adaptive adversary
ψ	(general) prophetic adversary
P	(general) randomized algorithm in dynamic network
Q	the randomized algorithm converted from P
I	input vector
C	coin flip outcomes in all rounds
$C_{[1:r]}$	coin flip outcomes in round 1 through r
C_r	coin flip outcomes in round r
$\tau(P, I, C)$	the dynamic network generated by τ under P, I, and C
$\text{cp}(P, I, \tau, C)$	communication pattern of P when running under I, τ, and C
$[r_1 : r_2]$	integers from r_1 (inclusive) to r_2 (inclusive)
$\text{tc}(P, I, \tau, C)$	time complexity of P when running under I, τ, and C
$\text{tc}_P(n, d)$	time complexity of P when running against adaptive adversaries with at most n nodes and at most d diameter
$\text{tc}^*_Q(n, d)$	time complexity of Q when running against prophetic adversaries with at most n nodes and at most d diameter
$\text{err}(P, I, \tau, C)$	error of P when running under I, τ, and C
$\text{err}_P(n)$	error of P when running against adaptive adversaries with at most n nodes and arbitrary diameter
$\text{err}^*_Q(n)$	error of Q when running against prophetic adversaries with at most n nodes and arbitrary diameter

Table 1: *Key notations.*

algorithms is perhaps no longer well-defined in dynamic networks, where a node's neighborhood changes over time. In comparison to these works, this paper considers i) *general* distributed algorithms that are not necessarily local, and ii) *dynamic networks* instead of static networks. Also because of this, our results and techniques are all quite different. For example, some of the key methods used in [7, 8, 10, 11, 15, 22] include network decomposition and conditional expectations, while we mainly rely on a novel simulation against a novel adaptive adversary.

2 Model

Table 1 summarizes our key notations.

Dynamic network and adversary. We consider a synchronous network with a fixed set of n nodes, where the nodes proceed in lock-step rounds. Throughout this paper, we assume that n is publicly known and that $n \geq 2$. All nodes start execution, simultaneously, from round 1. The nodes have unique ids from 0 through $n - 1$. Each node has some input value, and there is no constraint on the size of each input value. We will view the n input values as an input vector of length n. We consider *general distributed computing problems* where the n nodes aim to compute a certain function of the input vector. The output of the function is a

vector of length n, where node i should output the $(i+1)$-th entry in that vector. There is no constraint on the size of each output entry. An algorithm in this paper always refers to some algorithm for solving some problem that can be modeled as the above way.

The topology among the n nodes may change from round to round. Following [16, 18, 26], we assume that the topology is determined by some *adaptive adversary*. An *adaptive adversary* τ is an infinite sequence of functions $\tau_r(P, I, C_{[1:r]})$ for $r \geq 1$. Here τ_r takes as parameters the randomized algorithm P, the input vector I, and P's coin flip outcomes $C_{[1:r]}$ in round 1 (inclusive) to round r (inclusive). The function τ_r then outputs some connected and undirected graph with n nodes, as the topology of the network in round r. There is no other constraint on the graph. We also call this infinite sequence of graphs (starting from round 1) as a *dynamic network*, and say that τ is an adaptive adversary with n nodes. With a slight abuse of notation, we use $\tau(P, I, C)$ to denote the dynamic network produced by τ, under algorithm P, input vector I, and P's coin flip outcomes C across all rounds. (This is a slight abuse since τ is not a function.) For any given dynamic network $G = G_1 G_2 \ldots$ where G_r is the round-r topology of G, we define the special adaptive adversary γ^G to be $\gamma^G_r(P, I, C_{[1:r]}) = G_r$ for all r, P, I, and $C_{[1:r]}$. A *prophetic adversary* ψ is a function mapping the tuple (P, I, C) to a dynamic network $H = \psi(P, I, C)$. Since ψ is a single function (instead of a sequence of functions), ψ can see all coin flip outcomes of P in all rounds (i.e., C), before deciding the topology in each round.

We follow the communication model in [14, 26]: In each round, a node may choose to either send an $O(\log n)$ size message (i.e., the broadcast $\mathcal{CONGEST}$ model [24]) or to receive, as determined by the algorithm running on that node. A message sent in round r is received, by the end of round r, by all the receiving neighbors of the sender in round r.

Diameter. We adopt the standard notion of *dynamic diameter* [18] (or *diameter* in short) for dynamic networks. Formally, we define $(u, r) \to (v, r+1)$ if either $u = v$ or v is u's neighbor in round r. Let the relation "$\rightsquigarrow$" be the transitive closure of "$\to$". The *diameter* is defined as the smallest d such that $(u, r) \rightsquigarrow (v, r+d)$ holds for all u, v, and $r \geq 1$. Trivially, the diameter of a dynamic network with n nodes is at most $n-1$. Since the diameter is controlled by the adversary, it is not known to the algorithm beforehand. The *diameter* of an adaptive adversary τ (prophetic adversary ψ) is the smallest d where the diameter of $\tau(P, I, C)$ ($\psi(P, I, C)$) is at most d for all P, I, and C.

Time complexity and error. For the time complexity of an execution, we define the function $\text{tc}(P, I, \tau, C)$ to be the number of rounds needed for all nodes to output in P, when algorithm P runs with input vector I, adaptive adversary τ, and coin flip outcomes C. For the error of an execution, we define the binary function $\text{err}(P, I, \tau, C)$ to be 1 iff P's output is wrong (on any node), when P runs with I, τ, and C.

In the following, $\max_\tau$ will be taken over all adaptive adversaries τ with at most n nodes and at most d diameter. Unless otherwise specified, an algorithm in this paper can be either a Las Vegas (LV) algorithm or a Monte Carlo (MC) algorithm. We define a randomized algorithm P's *time complexity against adaptive adversaries* as $\text{tc}_P(n, d) = \max_I \max_\tau E_C[\text{tc}(P, I, \tau, C)]$ if P is an LV algorithm, or $\text{tc}_P(n, d) = \max_I \max_\tau \max_C \text{tc}(P, I, \tau, C)$ if P is an MC algorithm. We define an MC algorithm P's *error against adaptive adversaries* as $\text{err}_P(n) = \max_I \max_d \max_\tau E_C[\text{err}(P, I, \tau, C)]$.

We will need to reason about the properties of algorithm Q running against prophetic adversaries. Given Q's coin flip outcomes C in all rounds, since prophe-tic adversaries always see C beforehand, the worst-case prophetic adversary can always choose the worst-case dynamic network H for such C. Hence if Q is an LV algorithm, we define its *time complexity against prophetic adversaries* to be $\text{tc}^*_Q(n, d) = \max_I E_C[\max_H \text{tc}(Q, I, \gamma^H, C)]$. Note that here $\max_H$ is taken after C is given, and is taken over all dynamic networks with at most n nodes and at most d diameter. If Q is an MC algorithm, then its *time complexity / error against prophetic adversaries* will be $\text{tc}^*_Q(n, d) = \max_I \max_C \max_H \text{tc}(Q, I, \gamma^H, C)$

and $\text{err}^*_Q(n) = \max_I E_C[\max_d \max_H \text{err}(Q, I, \gamma^H, C)]$, respectively.

Conventions. All logarithms in this paper are base 2. We sometimes consider round 0 for convenience, where the algorithm does nothing and all nodes are receiving.

3 Adaptive Adversary Simulated by Q

As mentioned in Section 1, given some arbitrary randomized algorithm P, we want to construct algorithm Q that simulates the execution of P against some novel adaptive adversary $\alpha^{\epsilon,t}$. We want $\alpha^{\epsilon,t}$ to have small diameter so that P's time complexity (when running against $\alpha^{\epsilon,t}$) is small. Let H be the dynamic network over which Q runs. We further need Q to have good complexity and error guarantees, even if H is constructed by prophetic adversaries.

3.1 Intuition

Starting point. Recall that in any given round r, an adaptive adversary knows whether each node in P will be sending or receiving in that round (since the adaptive adversary sees C_r), before the adversary decides the topology in that round. Let us consider the following trivial topology as a starting point. In this topology, all nodes that are sending in the round form a clique, and all nodes that are receiving in the round form a clique. Some of the sending nodes will be chosen as *centers* for that round. A center will be connected to all other nodes (including all other centers) directly. To simulate P for one round over such a topology, we only need to deliver the message sent by each center to each of the receiving nodes. To do so, for each center, Q will flood the message (sent by the center) in the dynamic network H. Such flooding will obviously still work even if H is generated by a prophetic adversary. It takes total $d \cdot x$ rounds to simulate one round of P, where x is the number of centers and d is the diameter of H.

But there are several issues. Since only sending nodes can be centers and since a node may not always be sending in all rounds, we may be forced to keep switching the centers from round to round. This may then cause the (dynamic) diameter of the dynamic network to be large, despite the topology in each round having a small static diameter. One naive way to avoid this problem is to choose all sending nodes as centers. But doing so would result in too many centers, rendering the simulation inefficient. The following explains how we overcome these issues.

Choosing the centers. Our design of $\alpha^{\epsilon,t}$ uses only a logarithmic number of centers in each round. To obtain some intuition, consider any two consecutive rounds $r-1$ and r, where $r \geq 3$. We define $A^{RS}_r = \{u \mid u$ receives (hence the superscript "^{R}") in round $r-1$ and sends (hence the superscript "^{S}") in round $r\}$. Here "sends"/"receives" refers to u sending/receiving in the execution of P against $\alpha^{\epsilon,t}$. We similarly define the remaining three sets A^{SS}_r, A^{SR}_r, and A^{RR}_r. We hope to choose the centers in such a way that for some small d (e.g., $O(\log n)$), we have $(u, r-1) \rightsquigarrow (v, r+d-1)$ for all u and v. We will soon see that it will be convenient to consider u's in the 4 sets separately.

For round r, we will first pick some (arbitrary) node $w \in A^{RS}_r$ as a center. Such a center will ensure that for all $u \in A^{RS}_r$ and all v, we have $(u, r-1) \to (w, r) \to (v, r+1)$. Similarly, we will pick some (arbitrary) node $w \in A^{SS}_r$ as another center, to take care of $u \in A^{SS}_r$. Next, for any $u \in A^{SR}_r$, note that u must be in either A^{RS}_{r-1} or A^{SS}_{r-1}. If we chose the centers in round $r-1$ also according to the earlier rules, then such a u has already been taken care of as well.

The trickier case. The case for $u \in A^{RR}_r$ is trickier. In fact, to get some intuition, consider a node u that continuously receives in round 1 through round d. We want to ensure that $(u, 1) \rightsquigarrow (v, d+1)$ for all v. Let

S_r be the set of nodes that are sending in round r and let W_r be the set of centers in round r, for $1 \leq r \leq d$. For all $v \in \cup_r W_r$, we clearly have $(u, 1) \rightsquigarrow (v, d+1)$. For all $v \notin \cap_r S_r$, v must be receiving in some round $i \in [1, d]$, and hence we have $(u, i) \rightarrow (v, i+1)$ and we are done.

The case for $v \in (\cap_r S_r) \setminus (\cup_r W_r)$ is more complicated. Such a v has always been sending, but is never chosen as a center. Now consider such a v, and observe that if some center w in round r sends (again) in some round i where $r+1 \leq i \leq d$, then we must have $(u, 1) \rightsquigarrow (w, r) \rightsquigarrow (w, i) \rightarrow (v, i+1) \rightsquigarrow (v, d+1)$. Based on this observation, we will want to choose W_r from S_r such that some node in W_r will send in some round $i \geq r+1$. But whether a node sends in future rounds may depend on future coin flip outcomes of P, as well as the incoming messages in those rounds. An adaptive adversary (for P) does not have the incoming messages in future rounds. It is not supposed to see future coin flip outcomes either.

Our next observation is that the adaptive adversary in round r, before deciding the round-r topology, can actually determine the *probability* that a node u will be sending (again) in round $r+1$, *if the node u is currently already sending in round r*. The reason is that u will not receive any incoming messages in round r, no matter what the topology is. Hence the probability is uniquely determined by u's state at the beginning of round r. Now given such probabilities for all the nodes in S_r, when choosing the centers, we will choose those nodes from S_r whose probabilities (of sending in round $r+1$) are at least 0.5, and we call such nodes as *promising nodes*. If we include logarithmic number of promising nodes in W_r,then with good probability, there will exist some $w \in W_r$ that sends in round $r+1$. Due to some technicality, the number of promising nodes in W_r will actually need to increase with r, so that we can eventually take a union bound across even infinite number of rounds.

Finally, it is possible that we never have a sufficient number (i.e., logarithmic number) of promising nodes. In such a case, we will show that $(\cap_r S_r) \setminus (\cup_r W_r)$ will be empty with good probability.

3.2 Our Novel Adaptive Adversary $\alpha^{\epsilon,t}$

We now formally define $\alpha^{\epsilon,t}$ for $0 < \epsilon < 1$ and $t \geq 1$. The adaptive adversary $\alpha^{\epsilon,t}$ always generates a clique as the topology for round r when $r > t$. If $r \leq t$, then consider the given algorithm P, input vector I, and coin flip outcomes $C_{[1:r]}$. Based on C_r and the state of P at the beginning of round r, $\alpha^{\epsilon,t}$ can infer which nodes will be sending in round r, and which nodes are promising nodes. For all pairs of nodes u and v where either they are both sending in round r or they are both receiving in round r, the adversary $\alpha^{\epsilon,t}$ adds an undirected edge between them. Next $\alpha^{\epsilon,t}$ chooses up to $(2 \log \frac{2r}{\epsilon} + 3)$ nodes as *centers* for round r. For every center w and every node v, $\alpha^{\epsilon,t}$ adds an edge between w and v, if there is not already such an edge.

The centers are chosen in the following way. First, among all the nodes that were receiving in round $r-1$ and are sending in round r, if there are such nodes, choose the one with the smallest id as a center. Second, among all the nodes that were sending in round $r-1$ and are again sending in round r, if there are such nodes, choose the one with the smallest id as a center. Third, among all the nodes that were centers in round $r-1$ and are sending in round r, if there are such nodes, choose the one with the smallest id as a center. Finally, rank all the promising nodes in round r, by their ids from smallest to largest. Choose the first $2 \log \frac{2r}{\epsilon}$ nodes from this sequence as centers. If the sequence contains less than $2 \log \frac{2r}{\epsilon}$ nodes, choose all of them. Since these 4 criteria are not necessarily exclusive, a node may be chosen as a center multiple times.

One can easily verify that the topology generate by $\alpha^{\epsilon,t}$ in each round is always connected. We will be able to eventually prove (in Appendix D) that with probability at least $1-\epsilon$, the dynamic network generated by the adversary $\alpha^{\epsilon,t}$ has a diameter of at most $8 \log \frac{8tn}{\epsilon}$.

Algorithm 1 LV-P-Converted-To-Q().

/* This algorithm Q simulates P's execution against $\alpha^{\epsilon,t}$. For clarity, the pseudo-code does not explicitly include the input to Q (which is relayed to P). Without loss of generality, P's output on a node (when viewed as a numerical value) is assumed to be always non-negative. A node outputs only once in this algorithm. A node will suppress output if it previously has already outputted. */

1: $\epsilon \leftarrow 0.1$; $k \leftarrow 2$;
2: **repeat forever**
3: **forall** integers $d' \geq 1$ and $t \geq 2$ where i) d' and t are both powers of 2, ii) $d't \log t \leq k$, and iii) SimulateP() has not been previously executed for such d' and t in Step 5 **do**
4: $C^{d',t} \leftarrow$ fresh coin flips, for all rounds in P;
5: `return_v` $\leftarrow$ SimulateP(ϵ, d', t, $C^{d',t}$);
6: /* See Algorithm 2 for pseudo-code of SimulateP(). */
7: **if** (`return_v` ≥ 0) **then output** `return_v`;
8: **endforall**
9: $k \leftarrow 2k$;

4 Conversion from LV Algorithm P to LV Algorithm Q

4.1 Pseudo-code and Intuition

Overview. Given any LV algorithm P, our algorithm Q (pseudo-code in Algorithm 1) will simulate the execution of P against $\alpha^{\epsilon,t}$. (Effectively, Q will be simulating both P and the adversary $\alpha^{\epsilon,t}$.) We will ensure that Q works even against prophetic adversaries. In the following, a *simulated round* refers to one round of P in its simulated execution. Recall that in each simulated round, $\alpha^{\epsilon,t}$ chooses $O(\log \frac{r}{\epsilon})$ centers. For each center, Q will do a binary search (via logarithmic number of sequential floodings) to find the id of that center. For example, for the first center, Q will use a binary search to find the node with the smallest id, among all nodes that were receiving in the previous simulated round and are sending in the current simulated round. Next, for each center (which must be sending in P for the current simulated round), Q will determine the message it should send in P. Q will then flood this message, and then feed this message into all nodes that are receiving in P for the current simulated round.

Challenges and our solutions. A key difficulty in the above simulation is that Q does not know the diameter d of the dynamic network over which it runs. This means that Q does not know how long it takes for each flooding to complete. Of course, Q can naturally use the standard doubling-trick and maintains a guess d' for d. Recall that Q uses flooding i) for finding the centers via binary searches, and ii) for disseminating the messages of the centers. When $d' < d$, obviously both steps can be incorrect. We need to design Q so that it can deal with such incorrect behavior.

As a starting point, for each binary search, we have a designated node (node 0) flood for d' rounds its result of the binary search. If a node does not see such flooding from node 0, or if its binary search result is different, it knows that something is wrong and flags itself. Next for each center in this list, if it is not flagged, it will flood the message (that it should send in P) for d' rounds. Again, whoever not seeing this flooding will flag itself. In our design, once a node gets flagged, it will not participate in any of the flooding or binary search any more (for the current d' value), but will nevertheless spend the corresponding number of rounds doing nothing, so that it remains "in sync" with the non-flagged nodes.

At this point, we have three possibilities: i) $d' \geq d$ and no node gets flagged, ii) $d' < d$ and no node gets flagged, iii) $d' < d$ and some nodes get flagged. For the second case, because $d' < d$, it is not immediately clear what guarantees the simulation can offer — for example, whether the binary search still finds the

Algorithm 2 SimulateP(ϵ, d', t, $C^{d',t}$).

/* This subroutine simulates P's execution against $\alpha^{\epsilon,t}$ for t simulated rounds, while feeding coin flip outcomes $C^{d',t}$ into P, and while using d' as the guess for the diameter of the dynamic network over which Q runs. Without loss of generality, P's output (if viewed as a numerical value) is assumed to be non-negative. A node, once flagged, will do nothing in all steps except Step 19 and 23, but the node will still spend the same number of rounds to go through each step as other nodes. When the pseudo-code says a node u "floods" something for d' rounds, it means that the flooding originates from u, and all nodes in the system will spend exactly d' rounds participating in this flooding. */

1: `flagged` $\leftarrow false$; `return_v` $\leftarrow -1$;
2: **for** ($r \leftarrow 1$; $r \leq t$; $r \leftarrow r+1$) **do**
3: **if** I will send in the simulated round r of P **then**
4: simulate P's execution in round r using $C_r^{d',t}$;
5: `msg` $\leftarrow$ message sent by me in P;
6: **else**
7: `msg` $\leftarrow$ `m_bad`;
8: **endif**
9: $S \leftarrow \emptyset$; `center`[] $\leftarrow$ GetCenters(ϵ, r, d'); /* GetCenters() takes $\Theta(d' \log \frac{r}{\epsilon} \log n)$ rounds and returns a list of $2 \log \frac{2r}{\epsilon} + 3$ centers. See Algorithm 3 for pseudo-code. */
10: **for** each j where $1 \leq j \leq 2 \log \frac{2r}{\epsilon} + 3$ **do**
11: node 0 floods its `center`$[j]$ for d' rounds;
12: **if** (I do not receive anything in the flooding at Step 11) or (my `center`$[j]$ is different from what I received) **then** `flagged` $\leftarrow true$;
13: // At this point, the value of `center`$[j]$ must be the same on all non-flagged nodes.
14: **if** (`center`$[j] \neq \perp$) **then** the node corresponding to `center`$[j]$ floods its `msg` for d' rounds; **else** spend d' rounds doing nothing;
15: **if** (`center`$[j] \neq \perp$) and (I receive some message in the flooding at Step 14) and (the message received is not `m_bad`) **then** $S \leftarrow S \cup \{$message received$\}$;
16: **if** (`center`$[j] \neq \perp$) and (I receive either the message `m_bad` or no message in the flooding at Step 14) **then** `flagged` $\leftarrow true$;
17: **endfor**
18: **if** I will receive in the simulated round r of P **then** simulate P's execution in round r using $C_r^{d',t}$, with S being the set of received messages;
19: **if** (`flagged`) **then** send `m_flag`; **else** receive for 1 round;
20: **if** `m_flag` received **then** `flagged` $\leftarrow true$;
21: **if** P has output in simulated round r **then** `return_v` $\leftarrow$ P's output;
22: **endfor**
23: **if** (`flagged`) **then return** -2; **else return** `return_v`;

Algorithm 3 GetCenters(ϵ, r, d').

/* This subroutine returns an array of $2\log\frac{2r}{\epsilon}+3$ centers, some of which can be $\perp$. The centers are chosen according to the construction of $\alpha^{\epsilon,t}$ in Section 3.2. The subroutine does a binary search to find out the value for each entry in the array. It uses d' as the guess for the diameter of the dynamic network, and takes total $d'(2\log\frac{2r}{\epsilon}+3)\log(n+1)$ rounds. */

let `center[]` be an array of size $2\log\frac{2r}{\epsilon}+3$;
if (I was receiving in simulated round $(r-1)$ of P) and (I am sending in simulated round r of P) **then** $z \leftarrow$ my_id; **else** $z \leftarrow n$;
`center`$[1] \leftarrow$ FindMin(z, d'); /* See Algorithm 4 for pseudo-code of FindMin(). */
if (I was sending in simulated round $(r-1)$ of P) and (I am sending in simulated round r of P) **then** $z \leftarrow$ my_id; **else** $z \leftarrow n$;
`center`$[2] \leftarrow$ FindMin(z, d');
if (I was a center in simulated round $(r-1)$ of P) and (I am sending in simulated round r of P) **then** $z \leftarrow$ my_id; **else** $z \leftarrow n$;
`center`$[3] \leftarrow$ FindMin(z, d');
for ($i=4$; $i \leq 2\log\frac{2r}{\epsilon}+3$; $i \leftarrow i+1$) **do**
    if (I am a promising node in simulated round r of P) and (`center`$[j] \neq$ my_id for all $4 \leq j \leq i-1$) **then** $z \leftarrow$ my_id; **else** $z \leftarrow n$;
    `center`$[i] \leftarrow$ FindMin(z, d');
endfor
for all $1 \leq i \leq 2\log\frac{2r}{\epsilon}+3$ **if** `center`$[i] = n$ **then** `center`$[i] \leftarrow \perp$;
return `center`$[]$;

Algorithm 4 FindMin(z, d').

/* The input parameter z is an integer in $[0, n]$. This subroutine tries to use a binary search to find out the minimum input value among all nodes. It uses d' as the guess for the diameter of the dynamic network, and takes total $d'\log(n+1)$ rounds. */

let z's binary form be $b_1 b_2 \ldots b_{\log(n+1)}$, with b_1 being the most significant bit;
for ($s=1$; $s \leq \log(n+1)$; $s \leftarrow s+1$) **do**
    $x \leftarrow$ ExistValue(b_s, 0, d'); // See Algorithm 5 for pseudo-code of ExistValue().
    if (x) and ($b_s \neq 0$) **then** $b_{s'} \leftarrow 1$ for all $s' \geq s+1$ and $b_s \leftarrow 0$;
endfor
return $b_1 b_2 \ldots b_{\log(n+1)}$ as an integer;

Algorithm 5 ExistValue(z, x, d').

/* This subroutine tries to check whether any node in the dynamic network has invoked this subroutine with $z = x$. It uses d' as the guess for the diameter of the dynamic network, and takes total d' rounds. */

if ($z = x$) **then** `exist` $\leftarrow true$; **else** `exist` $\leftarrow false$;
repeat d' rounds
    if (`exist`) **then** send `m_exist`; **else** receive for 1 round;
    if I receive `m_exist` **then** `exist` $\leftarrow true$;
return `exist`;

smallest id. Fortunately, we will be able to prove that as long as no node gets flagged, the simulation is still "correct". Specifically, for disseminating the centers' messages, it is obvious that if no node gets flagged, then all nodes must have received those messages, regardless of whether $d' < d$. For the binary search part, we will be able to prove the following strong property: As long as the binary search returns the same value on all nodes (which is a necessary condition for no nodes being flagged), the result of the binary search must be correct, *even if* $d' < d$. Putting these together, this means that the second case still corresponds to a proper execution of P against $\alpha^{\epsilon,t}$.

The third case (i.e., $d' < d$ and some nodes get flagged) is trickier. The challenge is that the non-flagged nodes may think everything is fine and then happily generate a potentially wrong output. To deal with this, our design first lets the flagged nodes send a special message — whoever receives this message will get flagged as well. For each simulated round of P, our algorithm Q will allocate exactly one dedicated round in Q to do this.

Next, as a key technical step, we will be able to prove that with such a mechanism, somewhat interestingly, those non-flagged nodes actually still constitute *part* of a valid execution of P against some prophetic adversary ψ (but not a valid execution of P against $\alpha^{\epsilon,t}$). Our proof will explicitly construct this prophetic adversary ψ. Let G be the dynamic network generated by ψ. We will prove that G's topology is always connected in every round, while leveraging the fact that the topology of the dynamic network H (over which Q runs) is always connected. It is important to note that here we need to use a prophetic adversary (instead of an adaptive adversary) to generate G, since G depends on H, and since H is generated by some prophetic adversary.

To quickly summarize, we effectively have that i) if no node gets flagged, then Q must have properly simulated P's execution against $\alpha^{\epsilon,t}$, and ii) if some nodes get flagged, then Q (on the non-flagged nodes) must have properly simulated P's execution against some prophetic adversary ψ. Now since P is an LV algorithm, it will never have any error when running over any G, even if G is generated by a prophetic adversary. The reason is that G could also be generated by some adaptive adversary (e.g., by the adaptive adversary that always outputs G, regardless of P's inputs and P's coin flip outcomes), and P promises zero error under all adaptive adversaries. Thus the outputs of those non-flagged nodes will never be wrong, and can always be safely used. Of course, P's time complexity guarantee will no longer hold when running against ψ. But this will not cause any problem — if P takes too long to output, Q will increase d' and retry.

Using fresh coins. Finally, since we are using the doubling-trick to guess d already, we will use the same trick to guess the number of rounds needed for P to output. This will make our proof on Q a constructive proof, instead of an existential proof. It is worth mentioning that for each d' (the guess on d) and t (the guess on the number of simulated rounds needed for P to output), Q will simulate P using a fresh set of random coins. This is necessary because for a given set of coin flip outcomes, the adversary $\alpha^{\epsilon,t}$ may happen to have large diameter, causing P to take too many rounds to output. Finally, for each pair of d' and t values, the simulation of P takes about $d't \log t$ rounds. To make the guessing process efficient, we maintain a budget k that keeps doubling. For a given budget k, we simulate P for all (d', t) pairs where $d't \log t \leq k$ and that are constant factors apart from each other.

4.2 Final Results

Theorem 1 next states that Q's output will never be wrong. Its proof (in Appendix E) mainly relies on the intuition in the previous section. The proof is involved, because it is not sufficient to just consider whether a node is flagged at a certain time point in each simulated round — we actually consider two separate time points in each simulated round.

Theorem 1. *For any LV algorithm P, the output of Q (Algorithm 1) will never be wrong.*

Theorem 2 next (see Appendix F for proof) is our key result on Q's time complexity. It states that even when Q runs against a prophetic adversary, its time complexity is small. Furthermore, if P is *input-stable* (defined in Appendix C), then Q can be further derandomized into a deterministic algorithm without increasing Q's time complexity.

Theorem 2. *Let Q be Algorithm 1, and let P be any LV algorithm where $\Omega(1) \leq \mathrm{tc}_P(n,n) \leq O(n^{a_1})$ for some constant a_1.*

- *There exists constant a' (independent of n) such that for all d, we have $tc^*_Q(n,d) = \max_I E_C[\max_H \mathrm{tc}(Q,I,\gamma^H,C)] = d \cdot O(\log^3 n \times \mathrm{tc}_P(n, a' \log n))$, where $\max_H$ is taken over all dynamic networks H with at most n nodes and at most d diameter. Furthermore, if P is input-stable, then there exist some coin flip outcomes C^Q such that for all d, we have $\max_I \max_H \mathrm{tc}(Q,I,\gamma^H,C^Q) = d \cdot O(\log^3 n \times \mathrm{tc}_P(n, a' \log n))$.*
- *If $\mathrm{tc}_P(n,d) = \Theta(f(n) \cdot g(d))$ for some $f(n)$ and $g(d)$ where there exists some constant a_2 such that $\Omega(1) \leq f(n) \leq O(n^{a_2})$ and $\Omega(d) \leq g(d) \leq O(d^{a_2})$, then we have $\mathrm{tc}^*_Q(n,d) = O(\mathrm{polylog}(n)) \cdot \mathrm{tc}_P(n,d)$. Furthermore, if P is input-stable, then there exist some coin flip outcomes C^Q such that for all d, we have $\max_I \max_H \mathrm{tc}(Q,I,\gamma^H,C^Q) = O(\mathrm{polylog}(n)) \cdot \mathrm{tc}_P(n,d)$.*

To derandomize any given input-stable LV algorithm P, we combine Theorem 1 and 2 while plugging C^Q into Q. This gives a deterministic algorithm with the desired time complexity.

Acknowledgments

This work is partly supported by the grant MOE2017-T2-2-031 from Singapore Ministry of Education.

References

[1] M. Ahmadi, A. Ghodselahi, F. Kuhn, and A. Molla. The cost of global broadcast in dynamic radio networks. In *OPODIS*, 2015. 4

[2] M. Ahmadi, F. Kuhn, S. Kutten, A. R. Molla, and G. Pandurangan. The communication cost of information spreading in dynamic networks. In *ICDCS*, July 2019. 4

[3] J. Augustine, C. Avin, M. Liaee, G. Pandurangan, and R. Rajaraman. Information spreading in dynamic networks under oblivious adversaries. In *DISC*, 2016. 4

[4] C. Avin, M. Koucky, and Z. Lotker. How to explore a fast-changing world (cover time of a simple random walk on evolving graphs). In *ICALP*, July 2008. 1

[5] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson. On the power of randomization in online algorithms. *Algorithmica*, 11(1):2–14, Jan. 1994. 4

[6] A. Casteigts, P. Flocchini, W. Quattrociocchi, and M. Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):384–408, 2012. 1

[7] K. Censor-Hillel, M. Parter, and G. Schwartzman. Derandomizing local distributed algorithms under bandwidth restrictions. In *DISC*, Oct. 2017. 1, 4, 5

[8] Y. Chang, T. Kopelowitz, and S. Pettie. An exponential separation between randomized and deterministic complexity in the LOCAL model. In *FOCS*, 2016. 1, 4, 5

[9] C. Dutta, G. Pandurangan, R. Rajaraman, Z. Sun, and E. Viola. On the complexity of information spreading in dynamic networks. In *SODA*, Jan. 2013. 2, 4, 18

[10] L. Feuilloley and P. Fraigniaud. Randomized local network computing. In *SPAA*, 2015. 1, 4, 5

[11] M. Ghaffari, D. Harris, and F. Kuhn. On derandomizing local distributed algorithms. In *FOCS*, Oct. 2018. 1, 4, 5

[12] M. Ghaffari, N. Lynch, and C. Newport. The cost of radio network broadcast for different models of unreliable links. In *PODC*, July 2013. 4

[13] I. Jahja, H. Yu, and R. Hou. On the power of randomization in distributed algorithms in dynamic networks with adaptive adversaries. In *Proceedings of the 26th International European Conference on Parallel and Distributed Computing*, 2020. 1

[14] I. Jahja, H. Yu, and Y. Zhao. Some lower bounds in dynamic networks with oblivious adversaries. *Distributed Computing*, 33(1):1–40, Feb 2020. 2, 6

[15] K. Kawarabayashi and G. Schwartzman. Adapting local sequential algorithms to the distributed setting. In *DISC*, Oct. 2018. 4, 5

[16] F. Kuhn, N. Lynch, C. Newport, R. Oshman, and A. Richa. Broadcasting in unreliable radio networks. In *PODC*, July 2010. 2, 4, 6

[17] F. Kuhn, N. Lynch, and R. Oshman. Distributed computation in dynamic networks. In *STOC*, June 2010. 1, 2, 4, 18

[18] F. Kuhn, Y. Moses, and R. Oshman. Coordinated consensus in dynamic networks. In *PODC*, June 2011. 2, 4, 6

[19] F. Kuhn and R. Oshman. Dynamic networks: Models and algorithms. *SIGACT News*, 42(1):82–96, Mar. 2011. 1, 4

[20] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1996. 2

[21] M. Luby. Removing randomness in parallel computation without a processor penalty. *Journal of Computer and System Sciences*, 47(2):250–286, Oct. 1993. 1, 4

[22] M. Naor and L. Stockmeyer. What can be computed locally? *SIAM Journal on Computing*, 24(6):1259–1277, Dec. 1995. 1, 4, 5

[23] R. O'Dell and R. Wattenhofer. Information dissemination in highly dynamic graphs. In *Joint Workshop on Foundations of Mobile Computing (DIALM-POMC)*, Sept. 2005. 1

[24] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 1987. 2, 6

[25] S. Vadhan. Pseudorandomness. *Foundations and Trends in Theoretical Computer Science*, 7(1-3):1–336, 2012. 1, 4

[26] H. Yu, Y. Zhao, and I. Jahja. The Cost of Unknown Diameter in Dynamic Networks. *Journal of the ACM*, 65(5):31:1–31:34, Sept. 2018. 2, 6

A Roadmap

The first part of our appendix proves our main theorems for LV algorithms — Theorem 1 and Theorem 2. Specifically, Appendix B first proves some useful properties of SimulateP() in Algorithm 2. Next, Appendix C defines the notion of *input-stability*, and Appendix D proves the properties of the adaptive adversary $\alpha^{\epsilon,t}$ that Q needs to simulate. Finally, Appendix E and F prove Theorem 1 and 2, respectively.

The second part of our appendix proves our main theorem for MC algorithms — Theorem 14. Specifically, Appendix G first gives the intuitions and pseudo-code for converting any given MC algorithm P to another MC algorithm Q that has nice properties against prophetic adversaries. Appendix G then states and proves Theorem 14.

B Properties of SimulateP()

Theorem 1 and 2 are both about Algorithm 1 (i.e., the algorithm Q). Algorithm 1 invokes the subroutine SimulateP() in Algorithm 2. SimulateP(), in turn, invokes Algorithm 3 through Algorithm 5. At a high level, Algorithm 2 simulates P's execution against $\alpha^{\epsilon,t}$ for t simulated rounds — to do so, Algorithm 2 essentially simulates both the algorithm P and the adaptive adversary $\alpha^{\epsilon,t}$. Algorithm 3 obtains a certain number of centers, for the simulation of $\alpha^{\epsilon,t}$. Algorithm 4 uses a binary search to find out the minimum value among all nodes, while Algorithm 5 is a simple utility subroutine. In the next, we prove several useful technical lemmas from the pseudo-code.

Lemma 3 in the next proves the properties of Algorithm 4 (i.e., FindMin()). In this lemma, it is crucial (for our later proof) that even when $d' < d$, certain parts of the lemma still hold.

Lemma 3. *Consider any dynamic network H with n nodes and at most d diameter.*

- *For all i and d', if node i in H invokes FindMin(z_i, d') with some $z_i \in [0, n]$, then the invocation must return some integer value in $[0, z_i]$. (This is regardless of whether/which/when other nodes are invoking FindMin).*
- *Fix any d', and let every node i in H invoke FindMin(z_i, d') for some $z_i \in [0, n]$, simultaneously.*
  - *If FindMin() returns the same value on all nodes, then the value returned must be $\min_i z_i$.*
  - *If $d' \geq d$, then FindMin() must return the same value on all nodes.*

Proof.

- In Algorithm 4, the bits $b_1 b_2 \dots b_{\log(n+1)}$ start with a value of z_i. The only place where $b_1 b_2 \dots b_{\log(n+1)}$ can later get modified is at Step 4. Each such modification always changes b_s from 1 to 0, while further setting the remaining less significant bits (i.e., $b_{s+1} b_{s+2} \dots b_{\log(n+1)}$) to 1. Hence each such modification always decreases the value of the binary string $b_1 b_2 \dots b_{\log(n+1)}$. Finally, it is obvious that the return value must be a non-negative integer.
- For any value x, we will use $b_1^x b_2^x \dots$ to denote its binary form, with b_1^x being the most significant bit. Let j be any integer where $z_j = \min_i z_i$.
  - Let v be the common value returned by FindMin() on all nodes. Consider the condition at Step 4 of Algorithm 4. Let s_1 be the first s for which this condition is satisfied, during the execution of

FindMin(z_j, d') by node j. If such s_1 does not exist, then we must have $v = z_j = \min_i z_i$ and we are done. If such s_1 exists, the following will derive a contradiction.

We will have a number of properties by the definition of such s_1. First, we must have $b_1^v = b_1^{z_j}$, $b_2^v = b_2^{z_j}$, ..., $b_{s_1-1}^v = b_{s_1-1}^{z_j}$. Second, we must also have $b_{s_1}^{z_j} = 1$, $b_{s_1}^v = 0$. Finally, ExistValue(1, 0, d') must returned `true` on node j at Step 3 when $s = s_1$. This means that there must exist some node $k \neq j$, such that node k invoked ExistValue(0, 0, d') when node j invoked ExistValue(1, 0, d').

Since node k invoked ExistValue(0, 0, d') during its iteration for $s = s_1$ in Algorithm 4, we claim that during node k's execution of FindMin(z_k, d'), the condition at Step 4 of Algorithm 4 is never satisfied for any $s \leq s_1 - 1$. Otherwise on node k, the bit b_{s_1} would have been set to 1 at Step 4. From this claim, we further know that i) the value of $b_{s_1}^{z_k}$ must be 0, and ii) $b_1^v = b_1^{z_k}$, $b_2^v = b_2^{z_k}$, ..., $b_{s_1-1}^v = b_{s_1-1}^{z_k}$. Now we have $b_{s_1}^{z_j} = 1$, $b_{s_1}^{z_k} = 0$, and further $b_1^{z_j} = b_1^{z_k}$, $b_2^{z_j} = b_2^{z_k}$, ..., $b_{s_1-1}^{z_j} = b_{s_1-1}^{z_k}$. This implies that z_k is smaller than z_j, contradicting the fact that $z_j = \min_i z_i$.

– When $d' \geq d$, ExistValue(b_s, 0, d') at Step 3 must return the same value on all nodes, and the return value must be `true` if there exists some node invoking ExistValue(0, 0, d'). It can then be easily verified that FindMin() must return the same value on all nodes.

□

Lemma 4 in the next bounds the number of rounds incurred during an invocation of SimulateP():

Lemma 4. *For any constant $\epsilon \in (0, 1)$, there exists some constant a_3 such that for all $d' \geq 1$ and $t \geq 2$ that are both power of 2, $n \geq 2$, protocol P, and coin flip outcomes $C^{d',t}$, SimulateP(ϵ, d', t, $C^{d',t}$) takes at most $(d't \log t) \times (\frac{a_3}{2} \log n)$ rounds.*

Proof. Directly from the pseudo-code of SimulateP() in Algorithm 2. □

Lemma 5 in the next proves that when SimulateP() is invoked with a sufficiently large guess d' on the diameter, then no node will ever be flagged during that invocation:

Lemma 5. *Consider any given algorithm P, $\epsilon \in (0, 1)$, d', t, $C^{d',t}$, and dynamic network H with at most d' diameter. If all nodes in H invoke SimulateP(ϵ, d', t, $C^{d',t}$) simultaneously, then no node will ever get flagged (i.e., the `flagged` variable can never be true) during the execution of SimulateP(ϵ, d', t, $C^{d',t}$).*

Proof. First, the return values of GetCenters(ϵ, r, d') (at Step 9 of Algorithm 2) comes (indirectly) from various invocations of FindMin(). Since the diameter of H is at most d', Lemma 3 tells us that FindMin() must return the same value on all nodes. Hence all nodes will have the same return value from GetCenters(ϵ, r, d') at Step 9 of Algorithm 2, and the `center[]` array must be the same on all nodes. For any given non-$\perp$ entry in the array, the corresponding FindMin() invocation must have returned some value v where $v \leq n-1$ on all nodes. By Lemma 3 and since H's diameter is at most d', this implies that some node must have invoked FindMin(v, d'). By the pseudo-code in Algorithm 3, it is easy to verify that v must be the id of this node, and this node must be sending in the current simulated round of P. This in turn means that on this node, the condition at Step 3 of Algorithm 2 must have been satisfied, and hence `msg` $\neq$ `m_bad`.

With all the above properties, together with the fact that H's diameter is at most d', one can easily verify that the conditions at Step 12, Step 16, and Step 20 of Algorithm 2 will never be satisfied. Hence no node will ever get flagged during the execution of SimulateP(ϵ, d', t, $C^{d',t}$). □

Lemma 6 in the next proves that SimulateP() properly simulates the execution of P against $\alpha^{\epsilon,t}$, when no node gets flagged:

Lemma 6. *Consider any invocation of SimulateP(ϵ, d', t, $C^{d',t}$), by every node simultaneously, for any given algorithm P, $\epsilon \in (0,1)$, d', t, and $C^{d',t}$. If no node ever gets flagged (i.e., the* `flagged` *variable is never true) during such invocation, then the invocation must have* properly simulated *the first t rounds of P's execution under $\alpha^{\epsilon,t}$ and $C^{d',t}$, in the following sense: For all $1 \leq r \leq t$, the behavior of P in round r of its execution under $\alpha^{\epsilon,t}$ and $C^{d',t}$ is exactly the same as in Step 4 and 18 of SimulateP(). Furthermore, the centers selected by $\alpha^{\epsilon,t}$ in round r of the above execution are exactly the same as the centers selected at Step 9 of SimulateP().*

Proof. We will do a simple induction on r. Assume that the lemma holds for all rounds before round r, and we now consider round r.

If no node ever gets flagged, then immediately after Step 9 of SimulateP(), the `center[]` array on all nodes must be identical. Recall that each entry in the array is set to be the return value of an invocation of FindMin(). Consider any given entry in the array. Then for that entry, all the nodes must have the same return value from their respective invocations of FindMin(). Lemma 3 then tells us that such return value must be the minimum value across all nodes. Based on our inductive hypothesis and also the pseudo-code in Algorithm 3, one can easily verify that the entries in the `center[]` array must be exactly the same as the centers chosen by $\alpha^{\epsilon,t}$, if P runs against $\alpha^{\epsilon,t}$.

Furthermore, since no node ever gets flagged, it means that each node has received the message flooded by each center. Hence a node that is receiving in the simulated round can properly feed such a message as an incoming message into its simulation of P. The remainder of the proof easily follows from the pseudo-code of SimulateP() in Algorithm 2. □

C Definition of Input-stability

Theorem 2 states, among other things, that if the algorithm P is *input-stable*, then we can derandomize P (efficiently). Before giving a formal definition, we first give some intuition behind the notion of input-stability. Intuitively, P is considered input-stable if under any given dynamic network and any given coin flip outcomes, P's time complexity/error and communication pattern are independent of the input. Note that P's output and P's messages' contents can, and probably should, depend on the input. Under different dynamic networks and different coin flip outcomes, P can still have different time complexities/error and communication patterns. For example, consider the token dissemination problem [9, 17] where each node has a token as its input, and where we need all nodes to output all tokens. Here as long as P treats the tokens as opaque, P will be input-stable. Other the other hand, treating the tokens as opaque is not necessary for P to be input-stable.

The following gives the formal definition of *input-stability*. Given any algorithm P, input vector I, adaptive adversary τ, and coin flip outcomes C, we define P's *communication pattern* (denoted as $\text{cp}(P, I, \tau, C)$) to be a sequence of sets, where the rth set is the set of nodes that are sending in round r (for all $r \geq 1$) in P's execution under I, τ, and C. An algorithm P is *input-stable* if for all dynamic networks G, coin flip outcomes C, input vectors I_1 and I_2, the following holds: i) $\text{tc}(P, I_1, \gamma^G, C)/\text{tc}(P, I_2, \gamma^G, C) \leq 2$ (if P is an LV algorithm), ii) $\text{err}(P, I_1, \gamma^G, C) = \text{err}(P, I_2, \gamma^G, C)$ (if P is an MC algorithm), and iii) $\text{cp}(P, I_1, \gamma^G, C) = \text{cp}(P, I_2, \gamma^G, C)$. Several aspects of the definition are worth elaborating. First, the constant "2" in the first property above can be easily replaced by any constant larger than 1. Second, under

different G and C, an input-stable P can have different time complexities, error, and communication patterns. For example, a node may decide to send with a larger probability iff it has received a larger number of distinct messages so far in the execution, which depends on G. Finally, the equations do not need to hold for all adaptive adversaries τ — they only need to hold for γ^G.

D Properties of the Adaptive Adversaries $\alpha^{\epsilon,t}$ and $\beta^{\epsilon,t}$

The section proves various useful properties of the adaptive adversary $\alpha^{\epsilon,t}$. To facilitate reasoning, we will need to introduce another adaptive adversary $\beta^{\epsilon,t}$, as a stepping stone.

D.1 Diameters of $\alpha^{\epsilon,t}$ and $\beta^{\epsilon,t}$

Recall that we want $\alpha^{\epsilon,t}$ to have small diameter, so that the simulation can be efficient. To facilitate reasoning, we introduce a second adaptive adversary $\beta^{\epsilon,t}$. We will prove that $\beta^{\epsilon,t}$ always has $O(\log \frac{tn}{\epsilon})$ diameter, and that $\alpha^{\epsilon,t}$ generates the same dynamic network as $\beta^{\epsilon,t}$ with at least $1-\epsilon$ probability, with the probability being taken over the algorithm's coin flips. Our simulation will still simulate $\alpha^{\epsilon,t}$ instead of $\beta^{\epsilon,t}$. But we can easily draw a connection between the time complexity/error of P when running against $\alpha^{\epsilon,t}$ and when running against $\beta^{\epsilon,t}$.

Just drawing this connection alone is not yet enough. In the simulation, the algorithm Q will also need to efficiently determine, in a distributed fashion, whether $\alpha^{\epsilon,t}$ behaves exactly the same as $\beta^{\epsilon,t}$. In particular, Q should not be forced to directly check the diameter of the dynamic network generated by $\alpha^{\epsilon,t}$. To achieve this property, we first define the concept of $\alpha^{\epsilon,t}$ being *favorable*, and then define $\beta^{\epsilon,t}$. Being favorable will be a sufficient condition for $\alpha^{\epsilon,t}$ to behave the same as $\beta^{\epsilon,t}$. The algorithm Q will directly check whether $\alpha^{\epsilon,t}$ is favorable, which can be done efficiently.

In a dynamic network generated by $\alpha^{\epsilon,t}$, we say that a node is a *twice-center for round* r ($2 \leq r \leq t$) if it is a center in both round $r-1$ and round r. Round r *has a twice-center* if there exists some node that is twice-center for round r. In the next, we will use "rounds $[r_1 : r_2]$" to denote all rounds from round r_1 (inclusive) to round r_2 (inclusive).

Definition 7 ($\alpha^{\epsilon,t}$ being favorable)**.** *For any given algorithm P, input vector I, and coin flip outcomes C, we say that $\alpha^{\epsilon,t}$ is* favorable up to round r' *if for each r where $1 \leq r < r + 8\log\frac{4rn}{\epsilon} \leq \min(r', t)$, at least one of the following two properties holds in the dynamic network generated by $\alpha^{\epsilon,t}$ under P, I, and C: i) some round in rounds $[r+1 : r+8\log\frac{4rn}{\epsilon}]$ has a twice-center, or ii) no node in P sends continuously for $4\log\frac{4rn}{\epsilon}$ rounds in rounds $[r : r+8\log\frac{4rn}{\epsilon}]$.*

Definition 8 ($\beta^{\epsilon,t}$)**.** *In each round r, the adaptive adversary $\beta^{\epsilon,t}$ behaves exactly the same as $\alpha^{\epsilon,t}$ except that if $\alpha^{\epsilon,t}$ is not favorable up to round r (i.e., $\beta^{\epsilon,t}$ can examine the topologies generated by $\alpha^{\epsilon,t}$ in the first r rounds), then $\beta^{\epsilon,t}$ will use a clique as the topology in round r.*

One can easily verify that the topology generated by $\beta^{\epsilon,t}$ in each round is always connected. The next two theorems show that the diameter of $\beta^{\epsilon,t}$ is small, and that with probability at least $1-\epsilon$, $\alpha^{\epsilon,t}$ behaves the same as $\beta^{\epsilon,t}$.

Theorem 9. *For all $\epsilon \in (0,1)$, $t \geq 1$, algorithm P, input vector I, and coin flip outcomes C, the dynamic network $\beta^{\epsilon,t}(P,I,C)$ has a diameter of at most $8\log\frac{8tn}{\epsilon}$.*

Proof. Recall that "rounds $[r_1 : r_2]$" denotes all rounds from round r_1 (inclusive) to round r_2 (inclusive). We will also use $[r_1 : r_2]$ to denote all integers from r_1 (inclusive) to r_2 (inclusive). Define function $f(x) = 8 \log \frac{4xn}{\epsilon}$. Consider any round $r \geq 1$ and any two nodes u and v in the dynamic network. It suffice to prove that $(u, r) \rightsquigarrow (v, r + f(t) + 8)$. Recall that by the design of $\beta^{\epsilon,t}$, the topology in round $t + 1$ must be a clique. Hence the claim will trivially hold if $r + f(t) + 8 \geq t + 2$, and we will only prove for $r + f(t) \leq t - 7$. If $\alpha^{\epsilon,t}$ is not favorable up to round $r + f(t)$, then $\beta^{\epsilon,t}$ must have started using a clique as the topology, starting from round $r + f(t)$ or earlier. In such a case, we immediately have $(u, r) \rightsquigarrow (v, r + f(t) + 8)$.

The only remaining case is where $r + f(t) \leq t - 7$ and where $\alpha^{\epsilon,t}$ is favorable up to round $r + f(t)$. We trivially have $r \leq t$, $f(r) \leq f(t)$, and $r + f(r) \leq t - 7$. Since $f(r) \leq f(t)$ and since $\alpha^{\epsilon,t}$ is favorable up to round $r + f(t)$, $\alpha^{\epsilon,t}$ must be favorable up to round $r + f(r)$. By Definition 7, we know that at least one of the following two properties must hold:

- Some round in rounds $[r + 1 : r + f(r)]$ has a twice-center.
- No node sends continuously for $\frac{f(r)}{2}$ rounds in rounds $[r : r + f(r)]$.

If there exists a round $r_1 \in [r + 1 : r + f(r)]$ with a twice-center node w, then we directly have $(u, r) \rightsquigarrow (u, r_1 - 1) \to (w, r_1) \to (v, r_1 + 1) \rightsquigarrow (v, r + f(r) + 2) \rightsquigarrow (v, r + f(t) + 8)$, and we are done. If no node sends continuously for $\frac{f(r)}{2}$ rounds in rounds $[r : r + f(r)]$, then u must be receiving in some round r_2 where $r_2 \in [r : r + \frac{f(r)}{2} - 1]$, and v must be receiving in some round r_3 where $r_3 \in [r + \frac{f(r)}{2} : r + f(r)]$. We separately consider two possibilities in the following.

First, if u is always receiving in rounds $[r_2 + 1 : r + f(r)]$, then we have $(u, r) \rightsquigarrow (u, r_2) \rightsquigarrow (u, r_3) \to (v, r_3 + 1) \rightsquigarrow (v, r + f(r) + 2) \rightsquigarrow (v, r + f(t) + 8)$. Second, if u sends in some round within rounds $[r_2 + 1 : r + f(r)]$, let round r_4 be the earliest such round, where $r_2 + 1 \leq r_4 \leq r + f(r)$. This means that u receives in round $r_4 - 1$ and sends in round r_4. By the design of the adaptive adversary $\alpha^{\epsilon,t}$ (and hence $\beta^{\epsilon,t}$), we know that round r_4 must have a center w where w (w can be u itself) receives in round $r_4 - 1$ and sends in round r_4. Hence we have $(u, r) \rightsquigarrow (u, r_2) \rightsquigarrow (u, r_4 - 1) \to (w, r_4) \to (v, r_4 + 1) \rightsquigarrow (v, r + f(r) + 2) \rightsquigarrow (v, r + f(t) + 8)$. □

Theorem 10. *For any given $\epsilon \in (0, 1)$, $t \geq 1$, algorithm P, input vector I, define $A = \{C \mid \alpha^{\epsilon,t}$ is favorable up to round t for P, I, and C (which implies $\alpha^{\epsilon,t}(P, I, C) = \beta^{\epsilon,t}(P, I, C))\}$. Then A contains at least a $1 - \epsilon$ fraction of all possible coin flip outcomes.*

Proof. Consider any $r \geq 1$, we will prove that under any given P and I, $\alpha^{\epsilon,t}$ satisfies at least one of the two conditions in Definition 7 with probability at least $1 - \frac{\epsilon}{2r^2}$, with the probability taken over C. A union bound across all r will then show that $\alpha^{\epsilon,t}$ is favorable up to round t, with probability at least $1 - \sum_{r=1}^{\infty} \frac{\epsilon}{2r^2} > 1 - \epsilon$.

Recall that "rounds $[r_1 : r_2]$" denotes all rounds from round r_1 (inclusive) to round r_2 (inclusive), and that $[r_1 : r_2]$ denotes all integers from r_1 (inclusive) to r_2 (inclusive). Let $z = 8 \log \frac{4rn}{\epsilon}$. First, we consider the case where there exists some $r_1 \in [r : r + z - 1]$ such that round r_1 has at least $2 \log \frac{2r}{\epsilon}$ promising nodes. By the design of $\alpha^{\epsilon,t}$, at least $2 \log \frac{2r}{\epsilon}$ of these promising nodes will be centers in round r_1. With probability at least $1 - 0.5^{2 \log \frac{2r}{\epsilon}} > 1 - \frac{\epsilon}{2r^2}$, at least one of these centers will be sending again in round $r_1 + 1$. Let the non-empty set of such centers be W. Again by the design of $\alpha^{\epsilon,t}$, some node in W will become a center again in round $r_1 + 1$. Hence round $r_1 + 1$ will have a twice-center, hence satisfying the first property in Definition 7.

Next, we consider the case where every round in rounds $[r : r + z - 1]$ has less than $2 \log \frac{2r}{\epsilon}$ promising nodes. By the design of $\alpha^{\epsilon,t}$, in such a case all promising nodes will be centers. If there exists some

$r_2 \in [r : r + z - 1]$ such that some promising node in round r_2 sends (again) in round $r_2 + 1$, then round $r_2 + 1$ will have a twice-center node by the design of $\alpha^{\epsilon,t}$. This will then again satisfy the first property in Definition 7.

Now the only remaining case is that for every round in rounds $[r : r + z - 1]$, all promising nodes in that round will receive in the next round. Based on this, we will later prove $\Pr[A_1 \text{ and } A_2 \text{ and } A_3 \text{ and } A_4] \geq 1 - \frac{\epsilon}{2r^2}$, where A_i means that no node sends in every round in rounds $[r + \frac{i-1}{4}z : r + \frac{i}{4}z - 1]$ for $1 \leq i \leq 4$. Since $z \geq 16$, one can easily verify that every $\frac{z}{2}$ consecutive rounds in rounds $[r : r + z]$ must contain all the rounds in $[r + \frac{i-1}{4}z : r + \frac{i}{4}z - 1]$ for some i. This then implies that with probability at least $1 - \frac{\epsilon}{2r^2}$, no node sends continuously for $\frac{z}{2}$ rounds in rounds $[r : r + z]$, hence satisfying the second property in Definition 7.

To prove $\Pr[A_1 \text{ and } A_2 \text{ and } A_3 \text{ and } A_4] \geq 1 - \frac{\epsilon}{2r^2}$, it suffices to show that $\Pr[A_i] \geq 1 - \frac{\epsilon}{8r^2}$ for $1 \leq i \leq 4$. We only prove for $\Pr[A_1]$, since the other cases are similar. By our earlier argument, a promising node in any round $r_3 \in [r : r + \frac{z}{4} - 2]$ must be receiving in round $r_3 + 1$, and hence can never send in every round in rounds $[r : r + \frac{z}{4} - 1]$. Hence we only need to consider those nodes who are never promising nodes in any round in rounds $[r : r + \frac{z}{4} - 2]$. A non-promising node in a round will, with probability less that 0.5, send in the next round . Hence $\Pr[A_1] \geq 1 - n \times 0.5^{\frac{z}{4}-2} > 1 - \frac{\epsilon}{8r^2}$. □

D.2 Input-stability under $\alpha^{\epsilon,t}$ and $\beta^{\epsilon,t}$.

Given an input-stable (see Appendix C for definition) randomized algorithm P, our derandomization of P will need to reason about P's behavior under different inputs, when P runs against $\alpha^{\epsilon,t}$ and $\beta^{\epsilon,t}$. Obviously, we should leverage the fact that P is input-stable — namely, when running over a given dynamic network G, P's time complexity/error and communication pattern are independent of P's input. However, when P runs against $\alpha^{\epsilon,t}$ ($\beta^{\epsilon,t}$), it is not immediately clear whether these properties will continue to hold.

We will next prove that these properties will indeed continue to hold, when P runs against $\alpha^{\epsilon,t}$ ($\beta^{\epsilon,t}$). Intuitively, our proof relies on the following arguments. First, the round-r topology generated by $\alpha^{\epsilon,t}$ partly depends on which nodes are sending and which nodes are the promising nodes in round r. This in turn depends on the probabilities of certain nodes sending in round $r + 1$. We will be able to prove that these probabilities are independent of P's input, via an induction. Second, we will repeatedly invoke a simple indistinguishability argument: In round r, so far as P is concerned, $\alpha^{\epsilon,t}$ is indistinguishable from γ^G for some G, even though we do not know beforehand what G is. Formally, the following theorem states that if P runs against $\alpha^{\epsilon,t}$ ($\beta^{\epsilon,t}$), then P's time complexity/error, as well as the dynamic network generated by $\alpha^{\epsilon,t}$ ($\beta^{\epsilon,t}$), will not depend on P's input.

Theorem 11. *For all input-stable algorithm P, input vectors I_1 and I_2, coin flip outcomes C, and adaptive adversary $\tau \in \{\alpha^{\epsilon,t}, \beta^{\epsilon,t}\}$, we have:*

$$\begin{aligned} \tau(P, I_1, C) &= \tau(P, I_2, C), && \textit{for all } P \\ tc(P, I_1, \tau, C)/tc(P, I_2, \tau, C) &\leq 2, && \textit{if } P \textit{ is LV algorithm} \\ err(P, I_1, \tau, C) &= err(P, I_2, \tau, C), && \textit{if } P \textit{ is MC algorithm} \end{aligned}$$

Furthermore, for all $r \geq 1$, $\alpha^{\epsilon,t}$ is favorable up to round r for P, I_1, and C iff it is favorable up to round r for P, I_2, and C.

Proof. We first prove for $\tau = \alpha^{\epsilon,t}$. Note that $\alpha^{\epsilon,t}(P, I_1, C) = \alpha^{\epsilon,t}(P, I_2, C)$ directly follows from Lemma 12 (proved next). Let $G = \alpha^{\epsilon,t}(P, I_1, C)$. Since P is input-stable, if P is an LV algorithm, then

$\text{tc}(P, I_1, \alpha^{\epsilon,t}, C)/\text{tc}(P, I_2, \alpha^{\epsilon,t}, C) = \text{tc}(P, I_1, \gamma^G, C)/\text{tc}(P, I_2, \gamma^G, C) \leq 2$. If P is an MC algorithm, we have $\text{err}(P, I_1, \tau, C) = \text{err}(P, I_1, \gamma^G, C) = \text{err}(P, I_2, \gamma^G, C) = \text{err}(P, I_2, \tau, C)$.

Finally, Lemma 12 (proved next) implies that $\alpha^{\epsilon,t}$ is favorable (see Appendix D.1 for definition) up to round r for P, I_1, and C if and only if it is favorable up to round r for P, I_2, and C. Together with the fact that $\alpha^{\epsilon,t}(P, I_1, C) = \alpha^{\epsilon,t}(P, I_2, C)$, we have $\beta^{\epsilon,t}(P, I_1, C) = \beta^{\epsilon,t}(P, I_2, C)$.

The remaining case of $\tau = \beta^{\epsilon,t}$ is similar to the above proof for $\tau = \alpha^{\epsilon,t}$. □

Lemma 12. *Let P be any input-stable algorithm. Consider any input vectors I_1 and I_2, and coin flip outcomes C. Let G_r^1, S_r^1, and M_r^1 be the topology in round r, the set of nodes that are sending in round r, and the set of promising nodes in round r, respectively, when P runs under I_1, $\alpha^{\epsilon,t}$, and C. Define G_r^2, S_r^2, and M_r^2 similarly, under I_2 instead of I_1. Then we have $G_r^1 = G_r^2$, $S_r^1 = S_r^2$, and $M_r^1 = M_r^2$ for all r.*

Proof. We will use an induction on r. The induction base for $r = 0$ obviously holds. Assume that $G_i^1 = G_i^2$, $S_i^1 = S_i^2$, and $M_i^1 = M_i^2$, for all $0 \leq i \leq r-1$. Now consider round r:

- We first prove $S_r^1 = S_r^2$. Define dynamic network G such that in round i, G's topology is G_i^1 for $1 \leq i \leq r-1$ and is a clique for $i \geq r$. Let $\text{cp}_r(P, I, \tau, C)$ denote the set of nodes sending in round r when P runs under input I, adaptive adversary τ, and coin flip outcomes C. Note that given P, I, and C, the value of $\text{cp}_r(P, I, \tau, C)$ is uniquely determined by the topologies generated by τ in the first $r-1$ rounds. Combining with the fact the P is input-stable, we have $S_r^1 = \text{cp}_r(P, I_1, \alpha^{\epsilon,t}, C) = \text{cp}_r(P, I_1, \gamma^G, C) = \text{cp}_r(P, I_2, \gamma^G, C) = \text{cp}_r(P, I_2, \alpha^{\epsilon,t}, C) = S_r^2$.

- Given $S_r^1 = S_r^2$, we next prove $M_r^1 = M_r^2$. Consider any node $u \in S_r^1$. Whether $u \in M_r^1$ depends on whether in the execution of P under I_1 and $\alpha^{\epsilon,t}$, node u's probability of sending in round $r+1$ is at least 0.5. Note that when defining this probability, $C_{[1:r]}$ is already given, and the probability is defined over C_{r+1} (i.e., the coin flips in round $r+1$).

  To prove $(u \in M_r^1) \Leftrightarrow (u \in M_r^2)$, we will prove that in the execution of P under I_1 and $\alpha^{\epsilon,t}$, the probability of u sending in round $r+1$ is exactly the same as in the execution of P under I_2 and $\alpha^{\epsilon,t}$. To facilitate our following proof, under any given coin flip outcomes in the first j rounds, we will use the boolean function $s(j, I, \tau)$ to denote whether u will send in round j in the execution of P under I, τ, and those coin flip outcomes. Thus to prove that the probability of u sending in round $r+1$ is the same in the two executions, it suffice to show that under any given $C_{[1:r-1]}$, C_r, and C_{r+1}, $s(r+1, I_1, \alpha^{\epsilon,t}) = s(r+1, I_2, \alpha^{\epsilon,t})$.

  Consider any given $C_{[1:r-1]}$, and define the dynamic network G as defined earlier, where in the first $r-1$ rounds, G has the same topologies as the topology generated by $\alpha^{\epsilon,t}$ under the given $C_{[1:r-1]}$. Under the given $C_{[1:r-1]}$, any given C_r, and any given C_{r+1}, we claim that $s(r+1, I_1, \alpha^{\epsilon,t}) = s(r+1, I_1, \gamma^G)$. To see why, note that by the end of round $r-1$, u's behavior is exactly the same under $\alpha^{\epsilon,t}$ and under γ^G. Hence under the given $C_{[1:r-1]}$ and C_r, we have $s(r, I_1, \alpha^{\epsilon,t}) = s(r, I_1, \gamma^G) = true$ (since $u \in S_r^1$). Now since u is sending in round r, its behavior in round r is unaffected by the round-r topology of the dynamic network. In turn, u's behavior by the end of round r must be exactly the same under $\alpha^{\epsilon,t}$ and under γ^G. Hence under the given $C_{[1:r-1]}$, C_r, and C_{r+1}, we have $s(r+1, I_1, \alpha^{\epsilon,t}) = s(r+1, I_1, \gamma^G)$.

  By a similar argument, under the given $C_{[1:r-1]}$, any given C_r, and any given C_{r+1}, we have $s(r+1, I_2, \alpha^{\epsilon,t}) = s(r+1, I_2, \gamma^G)$. Finally, since P is input-stable, we must have $s(r+1, I_1, \gamma^G) = s(r+1, I_2, \gamma^G)$. This gives us $s(r+1, I_1, \alpha^{\epsilon,t}) = s(r+1, I_2, \alpha^{\epsilon,t})$.

- Finally we will prove $G_r^1 = G_r^2$. By the design of $\alpha^{\epsilon,t}$, one can easily verify that G_r^1 is a function of S_i^1 and M_i^1 ($1 \leq i \leq r$), while G_r^2 is a function of S_i^2 and M_i^2 ($1 \leq i \leq r$). Hence $G_r^1 = G_r^2$ immediately follows from $S_r^1 = S_r^2$, $M_r^1 = M_r^2$, and our inductive hypothesis.

□

E Proof for Theorem 1

With all the preparation in Appendix B through D, we next prove Theorem 1 (restated in the following):

Theorem 1. *For any LV algorithm P, the output of Q (Algorithm 1) will never be wrong.*

Proof. Consider any given invocation of SimulateP() at Step 5 of Algorithm 1 on any given node u. We will refer to the iteration with $r = i$ in SimulateP() (i.e., Step 3 to Step 21 in Algorithm 2) as round i of the *simulated execution*. Unless otherwise mentioned, all steps in the remainder of this proof refer to steps in Algorithm 2. We use $\texttt{flagged}_1(u,i)$ and $\texttt{flagged}_2(u,i)$ to denote that node u is already flagged by Step 19 and Step 21, respectively, in round i of the simulated execution. We use $\overline{\texttt{flagged}}_1(u,i)$ and $\overline{\texttt{flagged}}_2(u,i)$ to denote the negation of $\texttt{flagged}_1(u,i)$ and $\texttt{flagged}_2(u,i)$, respectively. We define that $\overline{\texttt{flagged}}_1(u,0)$ and $\overline{\texttt{flagged}}_2(u,0)$ holds for all u. We say that u *sends (receives) in round i of the simulated execution* if $\overline{\texttt{flagged}}_1(u,i)$ and if u satisfy the condition at Step 3 (Step 18). For all i, we define $W(i) = \{w \mid w$ is in the $\texttt{center}[]$ array on node 0 immediately after Step 9 in round i of the simulated execution$\}$, if $\overline{\texttt{flagged}}_2(0,i-1)$ holds. We define $W(i) = \emptyset$, if $\texttt{flagged}_2(0,i-1)$ holds.

Reference dynamic network. To facilitate proof, we will define a *reference dynamic network*: Let G_i be the topology of the reference dynamic network in round i, constructed in the following way. Consider any two nodes u and v. First, if $\texttt{flagged}_2(u,i)$ and $\texttt{flagged}_2(v,i)$, then there is an edge between them in G_i. Second, if $\overline{\texttt{flagged}}_1(u,i)$ and $\overline{\texttt{flagged}}_1(v,i)$, then there is an edge between them iff they either are both sending or are both receiving in round i of the simulated execution. Finally, there is an edge between u and v if $\overline{\texttt{flagged}}_1(u,i)$ and $v \in W(i)$. Note that these three cases are not necessarily mutually exclusive.

For all i, G_i is connected. We want to prove that G_i is connected for all i. Consider any given i. Define $F = \{u \mid \texttt{flagged}_1(u,i)\}$ and $\overline{F} = \{u \mid \overline{\texttt{flagged}}_1(u,i)\}$. Since $\texttt{flagged}_1(u,i)$ implies $\texttt{flagged}_2(u,i)$, F always forms a clique in G_i.

Next, if both F and $\overline{F}$ are non-empty, we claim that there must exists an edge in G_i connecting some node $u \in F$ to some node $v \in \overline{F}$. The reason is that all nodes in F send $\texttt{m\_flag}$ in Step 19, while all nodes in $\overline{F}$ receive in Step 19. Since the dynamic network over which Q runs is connected, in Step 20 there must be some node $v \in \overline{F}$ receives $\texttt{m\_flag}$ from some node $u \in F$. Then we will have $\texttt{flagged}_2(v,i)$, and by design of G_i, this will result in an edge between u and v (since $\texttt{flagged}_2(u,i)$ must hold as well).[3]

If $\overline{F} \neq \emptyset$, we will need to also reason about the nodes in $\overline{F}$. If all nodes in $\overline{F}$ receive in round i of the simulated execution, then $\overline{F}$ obviously also forms a clique in G_i and we are done.

The only remaining case, which is also the most complex case, is where $\overline{F} \neq \emptyset$ and some node $u \in \overline{F}$ sends in round i of the simulated execution. Consider the $\texttt{center}[]$ array on node u immediately after Step 9. We claim that the array must contain at least one non-$\perp$ entry. We will prove the claim for the case where u receives in round $i-1$ in the simulated execution — the case where u sends in round $i-1$ is rather similar. When u invokes GetCenters() at Step 9, it will invoke Algorithm 3. Now since u receives in round $i-1$

[3]Note that the dynamic network H over which Q runs determines which nodes are u and v. Hence if H is generated by a prophetic adversary (i.e., if H depends on Q's coin flip outcomes in future rounds), then G_i will depend on those coin flips outcomes as well. This is why we need a prophetic adversary ψ (instead of an adaptive adversary τ) to generate G_i.

and sends in round i, at Step 3 of Algorithm 3, it will invoke FindMin(z, d') with z being u's id. Lemma 3 then implies that on node u, FindMin(z, d') will return some value that is in $[0, n-1]$. This value will then become a non-$\perp$ entry in the `center[]` array on node u.

We have proved that the `center[]` array on node u immediately after Step 9 contains at least one non-$\perp$ entry. Note that $\overline{\texttt{flagged}}_1(u,i)$ must imply $\overline{\texttt{flagged}}_2(0,i-1)$. Also since $\overline{\texttt{flagged}}_1(u,i)$ holds, we know that immediately after Step 9, the `center[]` array on node 0 must be exactly the same as the `center[]` array on node u. In turn, by the definition of $W(i)$, we have $W(i) \neq \emptyset$. Now consider any $w \in W(i)$. All nodes in $\overline{F}$ will have any edge to w. Hence $\overline{F}$, together with w, must form a connected component in G_i. Putting everything together, we have shown that G_i is always connected.

Simulated execution = reference execution. Now consider the *reference execution* of P over the above reference dynamic network, where we feed the same input vector and coin flip outcomes into P as in the simulated execution. We will prove that for all node u, if $\overline{\texttt{flagged}}_2(u,i)$ holds, then in round i, node u's behavior in the reference execution and in the simulated execution must be exactly the same. We prove via an induction on i. Assume that the claim holds for round $i-1$. Now consider round i and any node u such that $\overline{\texttt{flagged}}_2(u,i)$ holds. If u is sending in round i of the reference execution, then u's behavior in round i of the reference execution is entirely determined by u's state at the end of round $i-1$ of the reference execution and the coin flip outcomes in round i. Applying our inductive hypothesis then immediately tells us that u's behavior in round i of the simulated execution must be the same as its behavior in the reference execution.

If u is receiving in round i of the reference execution, then applying our inductive hypothesis tells us that u must also receive in round i of the simulated execution. We will need to prove that the messages received in round i by u in the two executions are the same. Let $A_1 = \{v \mid$ in round i of the reference execution, v is sending and v is u's neighbor in $G_i\}$, and $A_2 = \{v \mid$ in round i of the simulated execution, v is sending and at Step 18 in Algorithm 2 on node u, the set S contains v's message$\}$.

We will first prove that $A_1 = A_2$. Consider any $v \in A_2$. In order for v's message to be included in the set S on u at Step 18, v must be in the `center[]` array on node u at Step 15.[4] Note that $\overline{\texttt{flagged}}_2(u,i)$ must imply $\overline{\texttt{flagged}}_2(0,i-1)$. Also because $\overline{\texttt{flagged}}_2(u,i)$ holds, we know that v must also be in the `center[]` array on node 0 immediately after Step 9, and hence $v \in W(i)$. Since $\overline{\texttt{flagged}}_2(u,i)$ implies $\overline{\texttt{flagged}}_1(u,i)$, together with $v \in W(i)$, we know that there is an edge in G_i connecting u and v. Hence $A_2 \subseteq A_1$.

Next consider any $v \in A_1$. We know that v is sending in round i, u is receiving in round i, and $\overline{\texttt{flagged}}_2(u,i)$ holds. Given how G_i is constructed, we know that either $u \in W(i)$ or $v \in W(i)$. If $u \in W(i)$, since $\overline{\texttt{flagged}}_2(u,i)$ holds, we know that u itself must be in the `center[]` array on node u immediately after Step 9. Since u is receiving in round i, at Step 14, u will flood its `msg` where `msg` = `m_bad` . This would imply $\texttt{flagged}_2(u,i)$, and hence it is impossible for $u \in W(i)$. Then the only remaining possibility is $v \in W(i)$. We claim that v must be in the `center[]` array on node u immediately after Step 9, and also that u must have included v's message in S at Step 18 — otherwise $\overline{\texttt{flagged}}_2(u,i)$ would not hold. This implies that $v \in A_2$ and hence $A_1 \subseteq A_2$.

So far we have proved that $A_1 = A_2$. Consider any $v \in A_2$. We need to further prove that in round i, the message sent by v in the reference execution is the same as the one in the simulated execution. We first claim that $\overline{\texttt{flagged}}_2(v,i-1)$ must hold. The reason is that otherwise in round i of the simulated execution, v would not flood its message and hence v would not belong to A_2. Now since $\overline{\texttt{flagged}}_2(v,i-1)$, we can invoke our inductive hypothesis, which shows that v's behavior must be the same in round $i-1$ in the two executions. Since v is sending in round i in both executions, and since a sending node's behavior is not

[4] Note that at Step 14 and 15, the value of `center`$[j]$ must be the same on all nodes that have not been flagged so far.

affected by the topology or the behavior of other nodes in that round, we know that v's behavior must be the same also in round i of the two executions. Hence v will send the same message in the two executions. In turn, the messages received in round i by node u in the two executions are the same.

Output of Q will never be wrong. We have just proved that as long as $\overline{\texttt{flagged}}_2(u,i)$ holds, u's behavior is the same in the two executions up to round i. In turn, this means that if u outputs in the simulated execution, it must also output the same value in the reference execution. Furthermore, in each round of the reference execution, the topology of the dynamic network is always connected. Since P is an LV protocol, all outputs in the reference execution (and hence in the simulated execution as well) must be correct. Thus Q's output will never be wrong. □

F Proof for Theorem 2

With all the preparation in Appendix B through D, we next prove Theorem 2 (restated in the following):

Theorem 2. *Let Q be Algorithm 1, and let P be any LV algorithm where $\Omega(1) \leq \text{tc}_P(n,n) \leq O(n^{a_1})$ for some constant a_1.*

- *There exists constant a' (independent of n) such that for all d, we have $tc^*_Q(n,d) = \max_I E_C[\max_H \text{tc}(Q,I,\gamma^H,C)] = d \cdot O(\log^3 n \times \text{tc}_P(n, a' \log n))$, where $\max_H$ is taken over all dynamic networks H with at most n nodes and at most d diameter. Furthermore, if P is input-stable, then there exist some coin flip outcomes C^Q such that for all d, we have $\max_I \max_H \text{tc}(Q,I,\gamma^H,C^Q) = d \cdot O(\log^3 n \times \text{tc}_P(n, a' \log n))$.*
- *If $\text{tc}_P(n,d) = \Theta(f(n) \cdot g(d))$ for some $f(n)$ and $g(d)$ where there exists some constant a_2 such that $\Omega(1) \leq f(n) \leq O(n^{a_2})$ and $\Omega(d) \leq g(d) \leq O(d^{a_2})$, then we have $\text{tc}^*_Q(n,d) = O(\text{polylog}(n)) \cdot \text{tc}_P(n,d)$. Furthermore, if P is input-stable, then there exist some coin flip outcomes C^Q such that for all d, we have $\max_I \max_H \text{tc}(Q,I,\gamma^H,C^Q) = O(\text{polylog}(n)) \cdot \text{tc}_P(n,d)$.*

Proof. Since $\text{tc}_P(n,n) = O(n^{a_1})$, there exists positive constant a_4 such that $tc_P(n,n) \leq n^{a_4}$ for all $n \geq 2$. Let $\epsilon = 0.1$, and let a' be any constant such that $8 \log \frac{160 a_4 n^{a_1+1}}{\epsilon} \leq a' \log n$ for all $n \geq 2$. Consider any given $n \geq 2$, and let $t_0 = 20\text{tc}_P(n, a' \log n) \leq 20 n^{a_4}$. We will only prove for the case where H has exactly n nodes and where both t_0 and d are power of 2 — generalizing to other cases is trivial. For any input vector I, define:

$$\begin{aligned} A_1(I) &= \{\text{coin flip outcomes } C^P \mid \alpha^{\epsilon,t_0}(P,I,C^P) = \beta^{\epsilon,t_0}(P,I,C^P)\} \\ A_2(I) &= \{\text{coin flip outcomes } C^P \mid \text{tc}(P,I,\beta^{\epsilon,t_0},C^P) \leq \frac{t_0}{2}\} \end{aligned}$$

Theorem 10 tells us that for all I, the set $A_1(I)$ contains at least 0.9 fraction of all coin flip outcomes. Next by Theorem 9, we have $E_{C^P}[\text{tc}(P,I,\beta^{\epsilon,t_0},C^P)] \leq \text{tc}_P(n, 8\log\frac{8t_0 n}{\epsilon}) \leq \text{tc}_P(n, 8\log\frac{160 n^{a_4+1}}{\epsilon}) \leq \text{tc}_P(n, a'\log n) = \frac{t_0}{20}$. Hence by Markov's inequality, we know that for all I, the set $A_2(I)$ contains at least 0.9 fraction of all coin flip outcomes. We next prove the various claims in the theorem one by one.

Proof for $\text{tc}^*_Q(n,d) = d \cdot O(\log^3 n \times \text{tc}_P(n, a' \log n))$. Consider any given I. We will prove that $E_C[\max_H \text{tc}(Q,I,\gamma^H,C)] = d \cdot O(\log^3 n \times tc_P(n, a' \log n))$ always hold. By the properties of $A_1(I)$ and $A_2(I)$, we know that for every $C^P \in A_1(I) \cap A_2(I)$, we have $\text{tc}(P,I,\alpha^{\epsilon,t_0},C^P) = \text{tc}(P,I,\beta^{\epsilon,t_0},C^P) \leq \frac{t_0}{2} \leq t_0$.

Throughout all invocations of SimulateP(ϵ, d', t_0, C^{d',t_0}) where $d' \geq d$, Lemma 5 tells us that no nodes will ever be flagged. In turn by Lemma 6, when running over any dynamic network H with at most d diameter, SimulateP(ϵ, d', t_0, C^{d',t_0}) must return P's output, if P would output within the first t_0 rounds when running under α^{ϵ,t_0} and C^{d',t_0}. Hence for all $C^{d',t_0} \in A_1(I) \cap A_2(I)$, all d, and all H with at most d diameter, SimulateP(ϵ, d', t_0, C^{d',t_0}) will return a non-negative value on all nodes when $d' \geq d$.

Recall that $A_1(I) \cap A_2(I)$ contains at least 0.8 fraction of all coin flip outcomes for P. Hence for all $d' \geq d$, with probability at least 0.8, SimulateP(ϵ, d', t_0, C^{d',t_0}) will return a non-negative value. Further note that for different d' (and hence different C^{d',t_0}), the probability of SimulateP(ϵ, d', t_0, C^{d',t_0}) returning a non-negative value is independent. Once SimulateP(ϵ, d', t_0, C^{d',t_0}) returns a non-negative value, Q will output at Step 7 of Algorithm 1. Putting everything together and by Lemma 13 (proved next), we have: $E_C[\max_H(\text{tc}(Q, I, \gamma^H, C))] \leq \sum_{i=0}^{\infty}(0.8 \times 0.2^i \times (2^i dt_0 \log t_0) \times (\log(2^i dt_0 \log t_0)) \times (a_3 \log n)) = d \cdot O(\log^3 n \times \text{tc}_P(n, a' \log n))$.

Proof for $\max_I \max_H \text{tc}(Q, I, \gamma^H, C^Q) = d \cdot O(\log^3 n \times \text{tc}_P(n, a' \log n))$**.** For this part, P is known to be input-stable. Consider any fixed input vector I_0. For all $C^P \in A_1(I_0) \cap A_2(I_0)$, we have $\text{tc}(P, I_0, \alpha^{\epsilon,t_0}, C^P) = \text{tc}(P, I_0, \beta^{\epsilon,t_0}, C^P) \leq \frac{t_0}{2}$. Theorem 11 then tells us that for all input vector I and all $C^P \in A_1(I_0) \cap A_2(I_0)$, we have $\text{tc}(P, I, \alpha^{\epsilon,t_0}, C^P) \leq 2\text{tc}(P, I_0, \alpha^{\epsilon,t_0}, C^P) \leq t_0$. Same as earlier, this then means that for all I, all dynamic networks H with at most d diameter, and all $C^{d',t_0} \in A_1(I_0) \cap A_2(I_0)$, SimulateP($\epsilon$, d', t_0, C^{d',t_0}) will always return a non-negative value on all nodes once d' reaches d. This will in turn cause Q to output at Step 7 of Algorithm 1.

Now let C^Q be any coin flip outcomes for Q, such that when Q generates C^{d',t_0} at Step 4 of Algorithm 1, we always have $C^{d',t_0} \in A_1(I_0) \cap A_2(I_0)$. Since $A_1(I_0) \cap A_2(I_0) \neq \emptyset$, such C^Q must exist. Lemma 13 (proved next) then tells us that under such C^Q and for some constant a_3, algorithm Q spends at most $(dt_0 \log t_0) \times (\log(dt_0 \log t_0)) \times (a_3 \log n)$ to finish executing SimulateP(ϵ, d', t_0, C^{d',t_0}) with $d' = d$ and some $C^{d',t_0} \in A_1(I_0) \cap A_2(I_0)$. Q will then immediately output. Hence we have $\max_I \max_H \text{tc}(Q, I, \gamma^H, C^Q) = d \cdot O(\log^3 n \times \text{tc}_P(n, a' \log n))$.

Proof for $\text{tc}^*_Q(n, d) = O(\text{polylog}(n)) \cdot \text{tc}_P(n, d)$**.** For this part, we have the condition that $tc_P(n, d) = \Theta(f(n) \cdot g(d))$ with $\Omega(1) \leq f(n) \leq O(n^{a_2})$ and $\Omega(d) \leq g(d) \leq O(d^{a_2})$. We already proved that $\text{tc}^*_Q(n, d) = d \cdot O(\log^3 n \times \text{tc}_P(n, a' \log n))$. With the new condition, we now have $\text{tc}^*_Q(n, d) = d \cdot O(\log^3 n \times f(n) \times g(a' \log n)) \leq g(d) \cdot O(\log^3 n \times f(n) \times (a' \log n)^{a_2}) = O(\text{polylog}(n)) \cdot f(n) \cdot g(d) = O(\text{polylog}(n)) \cdot \text{tc}_P(n, d)$.

Proof for $\max_I \max_H \text{tc}(Q, I, \gamma^H, C^Q) = O(\text{polylog}(n)) \cdot \text{tc}_P(n, d)$**.** The proof is similar to the above case and hence omitted to avoid redundancy. □

The following proves Lemma 13, which was used in the above proof, and which bounds the number of rounds needed for Q to finish executing SimulateP(ϵ, d', t, $C^{d',t}$):

Lemma 13. *Let Q be Algorithm 1. There exists some constant a_3 such that for all $d' \geq 1$ and $t \geq 2$ that are power of 2, all LV algorithm P, and all coin flip outcomes C for Q, by the end of the complete execution of SimulateP(ϵ, d', t, $C^{d',t}$) in Q, Q has spend at most $(d't \log t) \times (\log(d't \log t)) \times (a_3 \log n)$ rounds since the beginning of Q's execution.*

Proof. We use the a_3 from Lemma 4. It suffices to prove that by the time when Q completes its iteration (i.e., Step 3 to Step 8) for $k = d't \log t$, it has spent at most $(d't \log t) \times (\log(d't \log t)) \times (a_3 \log n)$ rounds since the beginning of its execution. For all i that is a power of 2, let t_i be the largest power of 2 such that $it_i \log t_i \leq k$. Let z_i be the total number of rounds needed to execute SimulateP(ϵ, i, t, $C^{i,t}$) for all t where t is a power of 2 and $it \log t \leq k$. By Lemma 4, we have $z_i \leq (i \times 2 \log 2) \times (\frac{a_3}{2} \log n) + (i \times 4 \log 4) \times$

Algorithm 6 MC-P-Converted-To-Q(ϵ).
/* All commentary notes under the title of Algorithm 1 apply to this algorithm as well. */

1: $k \leftarrow 2$; $C \leftarrow$ coin flips for all rounds in P;
2: **repeat forever**
3: **forall** $d' \geq 1$ and $t \geq 2$ where i) d' and t are both power of 2, ii) $d't \log t \leq k$, and iii) SimulateP() has not been previously executed for such d' and t in Step 4 **do**
4: `return_v` $\leftarrow$ SimulateP(ϵ, d', t, C); // See Algorithm 2 for pseudo-code of SimulateP().
5: `has_flag` $\leftarrow$ExistValue(`return_v`, $-2, n$); // See Algorithm 5 for pseudo-code of ExistValue().
6: `favorable` $\leftarrow$ CheckFavorable(ϵ, d', t); // See Algorithm 7 for pseudo-code of CheckFavorable().
7: **if** (!`has_flag`) and (!`favorable`) **then output** some arbitrary value;
8: **if** (!`has_flag`) and (`favorable`) and (`return_v` ≥ 0) **then output** `return_v`;
9: **endforall**
10: $k \leftarrow 2k$;

Algorithm 7 CheckFavorable(ϵ, d', t).
/* This subroutine tries to check whether the (simulated) dynamic network as generated by $\alpha^{\epsilon,t}$ is favorable. It uses d' as the guess for the diameter of the dynamic network over which Q runs, and takes total d' rounds. */

1: **if** there exists r where $1 \leq r < r + z \leq t$ and $z = 8 \log \frac{4rn}{\epsilon}$, such that (in rounds $[r + 1 : r + z]$, there does not exist a round where I have a twice-center in the `center`[] array in my invocation of SimulateP()) and (in rounds $[r : r + z]$, I sent continuously for $\frac{z}{2}$ rounds), **then** `favorable` $\leftarrow false$; **else** `favorable` $\leftarrow true$;
2: return (!ExistValue(`favorable`, $false, d'$)); /* See Algorithm 5 for pseudo-code of ExistValue(). */

$(\frac{a_3}{2} \log n) + \ldots + (i \times t_i \log t_i) \times (\frac{a_3}{2} \log n) < (i \times 2t_i \log t_i) \times (\frac{a_3}{2} \log n) \leq k \times (a_3 \log n)$. The total number of rounds needed for Q to completes its iteration (i.e., Step 3 to Step 8) for $k = d't \log t$ will be at most $z_1 + z_2 + z_4 + \ldots + z_k = (k \log k) \times (a_3 \log n)$. □

G Conversion from MC Algorithm P to MC Algorithm Q

So far in this paper, we have been focusing on converting any given LV algorithm P to another LV algorithm Q that has nice properties against prophetic adversaries. This section moves on to consider MC algorithms, and shows that we can also convert any given MC algorithm P to another MC algorithm Q that has nice properties against prophetic adversaries.

G.1 Pseudo-code and Intuitions

For any given MC algorithm P, Algorithm 6 and 7 give the pseudo-code for the corresponding MC algorithm Q. Our algorithm Q here follows the same overall framework as in Section 4.1, and the following discussion only focuses on the differences from Section 4.1.

As in Section 4.1, our algorithm Q here simulates P multiple times, for different values of d' (i.e., guess on diameter) and t (i.e., guess on the number of simulated rounds needed for P to output). Section 4.1 used fresh coin flips for each such simulation. But here since P is an MC algorithm, using fresh coin flips would amplify P's error excessively. Hence here Q will have to feed the same coin flip outcomes C into all the simulations. Now if $\alpha^{\epsilon,t}$ happens to be not favorable under the given C, it may generate a dynamic network with large diameter, causing P (and in turn Q) to take too many rounds to output. To avoid this problem, we will have Q explicitly check whether $\alpha^{\epsilon,t}$ is favorable, in a distributed fashion. If Q finds $\alpha^{\epsilon,t}$ to be not favorable, Q will generate some arbitrary output immediately. For checking whether $\alpha^{\epsilon,t}$ is favorable, Q

will also use the guess d' as the diameter of the dynamic network over which Q runs. If d' is no smaller than the actual diameter, the checking will be error-free. Otherwise the checking may have one-sided error — interestingly, our proof will still go through despite such error.

When some nodes are flagged, Section 4.1 showed that on all the non-flagged nodes, Q has nevertheless still properly simulated P's execution against some prophetic adversary ψ. This trick unfortunately no longer works for an MC algorithm P, because when running against the prophetic adversary ψ, P's error guarantee no longer holds. Thus here, after each simulation of P (for any d' and t values), we will have Q explicitly check whether there are any flagged nodes in the system. Each such checking takes $\Theta(n)$ rounds. Hence our final result for MC algorithms will have an extra additive $O(n \log^2 n)$ term.

G.2 Final Results

The following theorem is our final result for MC algorithms:

Theorem 14. *Let P be any MC algorithm where for some constants a_1 and δ, we have $\Omega(1) \leq \text{tc}_P(n,n) \leq O(n^{a_1})$ and $\text{err}_P(n) \leq \delta < 1$. Let Q be Algorithm 6 with constant $\epsilon \in (0, 1-\delta)$.*

- *There exists constant a' (independent of n) such that for all d, we have $\text{tc}^*_Q(n,d) = \max_I \max_C \max_H \text{tc}(Q, I, \gamma^H, C) = d \cdot O(\log^3 n \times \text{tc}_P(n, a' \log n)) + O(n \log^2 n)$. Here $\max_H$ is taken over all dynamic networks H with at most n nodes and at most d diameter. Furthermore, if $\text{tc}_P(n,d) = \Theta(f(n) \cdot g(d))$ for some $f(n)$ and $g(d)$ where there exists some constant a_2 such that $\Omega(n) \leq f(n) \leq O(n^{a_2})$ and $\Omega(d) \leq g(d) \leq O(d^{a_2})$, then $\text{tc}^*_Q(n,d) = O(\text{polylog}(n)) \cdot \text{tc}_P(n,d)$.*
- *For all n, we have $\text{err}^*_Q(n) \leq \delta + \epsilon$. Furthermore, if P is input-stable, then there exist coin flip outcomes C^Q such that $\max_I \max_H \text{err}(Q, I, \gamma^H, C^Q) = 0$, with $\max_H$ being taken over all dynamic networks H with at most n nodes.*

The above theorem also captures our derandomization result: To derandomize any given input-stable MC algorithm P, we simply take $\epsilon \leftarrow \frac{1-\delta}{2}$ and then plug C^Q into Q in the above theorem. We will then get an algorithm Q with no error and with the desired time complexity. Theorem 14 directly follows from Theorem 15 and Theorem 16, which we state and prove in the next.

Theorem 15. *Let P be any MC algorithm where for some constant a_1 and δ, we have $\Omega(1) \leq \text{tc}_P(n,n) \leq O(n^{a_1})$ and $\text{err}_P(n) \leq \delta < 1$. Let Q be Algorithm 6 with constant $\epsilon \in (0, 1-\delta)$. Then there exists constant a' (independent of n) such that for all d, we have $\text{tc}^*_Q(n,d) = d \cdot O(\log^3 n \times \text{tc}_P(n, a' \log n)) + O(n \log^2 n)$. Here $\max_H$ is taken over all dynamic networks H with at most n nodes and at most d diameter. Furthermore, if $\text{tc}_P(n,d) = \Theta(f(n) \cdot g(d))$ for some $f(n)$ and $g(d)$ where there exists some constant a_2 such that $\Omega(n) \leq f(n) \leq O(n^{a_2})$ and $\Omega(d) \leq g(d) \leq O(d^{a_2})$, then $\text{tc}^*_Q(n,d) = O(\text{polylog}(n)) \cdot \text{tc}_P(n,d)$.*

Proof. Recall that $\text{tc}^*_Q(n,d) = \max_I \max_C \max_H \text{tc}(Q, I, \gamma^H, C)$. Hence to prove the first part of the theorem, we only need to show that for any given I, C, and H, we always have $\text{tc}(Q, I, \gamma^H, C) \leq a_5 d \log^3 n \times \text{tc}_P(n, a' \log n) + a_5 n \log^2 n$ for some universal constant a_5. We only prove the case where H has exactly n nodes and where H's diameter d is a power of 2 — generalizing to other cases is trivial.

Since $\text{tc}_P(n,n) = O(n^{a_1})$, there exists positive constant a_4 such that $tc_P(n,n) \leq n^{a_4}$ for all n. Let constant a' be such that $8 \log \frac{8n^{a_4+1}}{\epsilon} < a' \log n$ for all $n \geq 2$. Let $t_0 = \text{tc}_P(n, a' \log n) \leq \text{tc}_P(n,n) \leq n^{a_4}$. We first prove that Q must output by the time that Q executes Step 8 in Algorithm 6 after it completes the execution of SimulateP(ϵ, d, t_0, C).

Consider the iteration (i.e., Step 4 to Step 8) in Algorithm 6 during which SimulateP(ϵ, d, t_0, C) is invoked. Since H has a diameter of at most d, Lemma 5 tells us that throughout the execution of SimulateP(ϵ, d, t_0, C), no node ever gets flagged. In turn by Lemma 6, SimulateP(ϵ, d, t_0, C) must have properly simulated the execution of P against the adversary α^{ϵ,t_0} up to round t_0. Furthermore, since no node gets flagged, we must have `return_v` ≥ -1 on all nodes, and hence ExistValue(`return_v`, -2, n) must return `false` on all nodes. This means that `has_flag` $= false$ on all nodes. Next, if on any node `favorable` $= false$ immediately after Step 6, then on that node Q must output at Step 7. Now consider any node u where `favorable` $= true$ immediately after Step 6 on u. Then CheckFavorable(ϵ, d, t_0) must have returned $true$ on u. Since i) the diameter of the dynamic network H is at most d, ii) CheckFavorable(ϵ, d, t_0) returned `true` on u, and iii) SimulateP(ϵ, d, t_0, C) has properly simulated the execution of P against α^{ϵ,t_0} up to round t_0, one can easily verify that α^{ϵ,t_0} is favorable up to round t_0 for the given P, I, and C. We hence have $\alpha^{\epsilon,t_0}(P, I, C) = \beta^{\epsilon,t_0}(P, I, C)$, by the construction of β^{ϵ,t_0} (see Definition 8 in Appendix D.1 for β^{ϵ,t_0}). By Theorem 9, the adversary β^{ϵ,t_0} has a diameter of at most $8 \log \frac{8t_0 n}{\epsilon} \leq 8 \log \frac{8n^{a_4+1}}{\epsilon} < a' \log n$. Hence we have $\text{tc}(P, I, \alpha^{\epsilon,t_0}, C) = \text{tc}(P, I, \beta^{\epsilon,t_0}, C) \leq \text{tc}_P(n, a' \log n) = t_0$. Against because SimulateP(ϵ, d, t_0, C) has properly simulated the execution of P against α^{ϵ,t_0} up to round t_0, this means that on node u, the condition at Step 21 of Algorithm 2 must have been satisfied at least once during the execution of SimulateP(ϵ, d, t_0, C). In turn, this means that on node u, SimulateP(ϵ, d, t_0, C) must return some non-negative value, and Q will then output at Step 8.

We have proved that Q must output by the time that Q executes Step 8 in Algorithm 6 after it completes the execution of SimulateP(ϵ, d, t_0, C). We next reason about the number of rounds needed for Q to complete the execution of SimulateP(ϵ, d, t_0, C) as well as Step 5 to Step 8 in Algorithm 6 after that.

Let $k = dt_0 \log t_0$. For all i that is a power of 2, let t_i be the largest power of 2 such that $it_i \log t_i \leq k$. Note that since $k \leq n \cdot n^{a_4} \log n^{a_4}$, we must have $t_i \leq n^{a_4+1}$ and $\log t_i \leq (a_4 + 1) \log n$. Let z_i be the total number of rounds needed to complete the execution of SimulateP(ϵ, i, t, C) as well as Step 5 to Step 8 after that, for all t where t is a power of 2 and where $it \log t \leq k$. By Lemma 4, for some constant a_3, we have $z_i \leq ((i \times 2 \log 2) \times (\frac{a_3}{2} \log n) + n + i) + ((i \times 4 \log 4) \times (\frac{a_3}{2} \log n) + n + i) + \ldots + ((i \times t_i \log t_i) \times (\frac{a_3}{2} \log n) + n + i) < (i \times 2t_i \log t_i) \times (\frac{a_3}{2} \log n) + (n + i) \log t_i \leq a_3 k \log n + n \log t_i + k \leq (a_3 \log n + 1)k + (a_4 + 1)n \log n$. The total number of rounds needed to complete the execution of SimulateP(ϵ, d, t_0, C), as well as Step 5 to Step 8 in Algorithm 6 after that, will be at most $z_1 + z_2 + z_4 + \ldots + z_k = (k \log k) \times (a_3 \log n + 1) + (a_1 + 1)n \log k \log n = dt_0 \log t_0 \times \log(dt_0 \log t_0) \times (a_3 \log n + 1) + (a_1 + 1)n \log(dt_0 \log t_0) \times \log n \leq d \times \text{tc}_P(n, a' \log n) \times \log n^{a_4} \times \log(n \cdot n^{a_4} \cdot \log n^{a_4}) \times (a_3 \log n + 1) + (a_1 + 1)n \log(n \cdot n^{a_4} \cdot \log n^{a_4}) \times \log n \leq a_5 d \log^3 n \times \text{tc}_P(n, a' \log n) + a_5 n \log^2 n$ for some universal constant a_5.

We have finished proving that $\text{tc}^*_Q(n, d) = d \cdot O(\log^3 n \times tc_P(n, a' \log n)) + O(n \log^2 n)$. Finally, if $\text{tc}_P(n, d) = \Theta(f(n) \cdot g(d))$, we will have:

$$\begin{aligned}
\text{tc}^*_Q(n, d) &= d \cdot O(\log^3 n \times f(n) \times g(a' \log n)) + O(n \log^2 n) \\
&\leq g(d) \cdot O(\log^3 n \times f(n)(a' \log n)^{a_2}) + f(n) O(\log^2 n) \\
&= O(\text{polylog}(n)) \cdot g(d) \cdot f(n) + O(\text{polylog}(n)) \cdot f(n) \\
&= O(\text{polylog}(n)) \cdot \text{tc}_P(n, d)
\end{aligned}$$

□

Theorem 16. *Let P be any MC algorithm where for some constant a_1 and δ, we have $\Omega(1) \leq \text{tc}_P(n, n) \leq O(n^{a_1})$ and $\text{err}_P(n) \leq \delta < 1$. Let Q be Algorithm 6 with constant $\epsilon \in (0, 1 - \delta)$. Then for all n, we have $\text{err}^*_Q(n) \leq \delta + \epsilon$. Furthermore, if P is input-stable, then there exist coin flip outcomes C^Q such that*

$\max_I \max_H \text{err}(Q, I, \gamma^H, C^Q) = 0$, *with* $\max_H$ *being taken over all dynamic networks* H *with at most* n *nodes.*

Proof. Theorem 15 showed that for any given n and for all d, all nodes running Q must output within some fixed number of rounds. Within these rounds, a node can only invoke SimulateP(ϵ, d', t, C) at Step 4 of Algorithm 6 for finite number of times. Let t_0 be the largest t such that before all nodes output in Q, some node has invoked SimulateP(ϵ, d', t, C) for some ϵ, d', and C.

Definitions and properties of A_1 and A_2. For the first part of the theorem, recall that $\text{err}^*_Q(n) = \max_I E_C[\max_d \max_H \text{err}(Q, I, \gamma^H, C)]$. Hence we need to prove that for any give I, we have $E_C[\max_d \max_H \text{err}(Q, I, \gamma^H, C)] \leq \delta + \epsilon$. For any given I, define:

$$\begin{aligned} A_1(I) &= \{\text{coin flip outcomes } C \mid \alpha^{\epsilon,t_0} \text{ is favorable up to round } t_0 \text{ for } P, I, \text{ and } C\} \\ A_2(I) &= \{\text{coin flip outcomes } C \mid \text{err}(P, I, \beta^{\epsilon,t_0}, C) = 0\} \end{aligned}$$

Since $\text{err}_n(P) \leq \delta$, the set $A_2(I)$ must contain at least a $1-\delta$ fraction of all possible coin flip outcomes. By Theorem 10, the set $A_1(I)$ contains at least a $1-\epsilon$ fraction of all possible coin flip outcomes. For all $C \in A_1(I) \cap A_2(I)$, we immediately have $\alpha^{\epsilon,t_0}(P, I, C) = \beta^{\epsilon,t_0}(P, I, C)$ and $\text{err}(P, I, \alpha^{\epsilon,t_0}, C) = \text{err}(P, I, \beta^{\epsilon,t_0}, C) = 0$. By the design of $\alpha^{\epsilon,t}$, we further know that for all $C \in A_1(I) \cap A_2(I)$ and $t \leq t_0$:

- The topologies generated (and the centers chosen) by $\alpha^{\epsilon,t}$ in the first t rounds is exactly the same as the topologies generated (and the centers chosen) by α^{ϵ,t_0} in those rounds.
- The adversary $\alpha^{\epsilon,t}$ is favorable up to round t for P, I, and C.
- When P runs against $\alpha^{\epsilon,t}$, it will never generate a wrong output within the first t rounds. (Otherwise P would generate the same wrong output within the first t rounds when running against α^{ϵ,t_0}, causing $\text{err}(P, I, \alpha^{\epsilon,t_0}, C)$ to be 1.)

Outputting at Step 7 and 8. Now consider any given input vector I, any given $C \in A_1(I) \cap A_2(I)$, any given node u, and any given iteration (i.e., Step 4 to Step 8) in Algorithm 6 on node u, during which SimulateP(ϵ, d', t, C) is invoked. Note that u may generate an output at either Step 7 or Step 8. We consider the value of `has_flag` on u, at Step 7 and Step 8. If `has_flag` $= true$, then u will output in neither Step 7 nor Step 8.

If `has_flag` $= false$, then ExistValue(`return_v`, -2, n) must have returned $false$ on node u. Because the diameter of the dynamic network H can be at most n (since the topology of H is a connected graph in each round), we know that no node invoked ExistValue(-2, -2, n) in this iteration — otherwise the invocation of ExistValue(`return_v`, -2, n) on node u would have returned $true$. This then implies that on every node, SimulateP(ϵ, d', t, C) returned some value that is larger than -2. Thus no node ever got flagged during its respective execution of SimulateP(ϵ, d', t, C). By Lemma 6, we now know that SimulateP(ϵ, d', t, C) must have properly simulated the first t rounds of P's execution under $\alpha^{\epsilon,t}$ and C. In particular, the centers selected by $\alpha^{\epsilon,t}$ in round r ($1 \leq r \leq t$) must be exactly the same as the centers selected at Step 9 of Algorithm 2.

By our earlier argument, since $C \in A_1(I) \cap A_2(I)$, we know that $\alpha^{\epsilon,t}$ is favorable up to round t for P, I, and C. Together with the fact that the centers selected by $\alpha^{\epsilon,t}$ in round r ($1 \leq r \leq t$) must be exactly the same as the centers selected at Step 9 of Algorithm 2, we know that the condition at Step 1 of Algorithm 7 will never be satisfied on any node. One can then verify that, *regardless of the relation between* d' *and* d, CheckFavorable(t, d') must return `true` on all nodes. This means that on node u, we will have `favorable` $= true$ at Step 7 of Algorithm 6. Hence u will not output at that step.

Next also by our earlier argument, since $C \in A_1(I) \cap A_2(I)$, we know that P never generates a wrong output in the first t rounds when running under I, $\alpha^{\epsilon,t}$, and C. Now since SimulateP(ϵ, d', t, C) has properly simulated the first t rounds of P's execution under $\alpha^{\epsilon,t}$ and C, if SimulateP(ϵ, d', t, C) returns a non-negative value, that value must be the correct output for the problem. Hence u will not generate a wrong output at Step 8 of Algorithm 6.

Putting it altogether. We have proved that under any given input vector I and any given $C \in A_1(I) \cap A_2(I)$, a node u (running Q) will generate a wrong output neither at Step 7 nor at Step 8. Since these two steps are the only steps in Algorithm 6 where u may output, and since $A_1(I) \cap A_2(I)$ contains at least a $1 - \epsilon - \delta$ fraction of all possible coin flip outcomes, we must have $E_C[\max_d \max_H \text{err}(Q, I, \gamma^H, C)] \leq \delta + \epsilon$ for the given I.

For input-stable P. We next prove the second part of the theorem, which is for input-stable P. Consider any fixed input vector I_0 and let C^Q be any given coin flip outcomes in $A_1(I_0) \cap A_2(I_0)$. Our proof above has already shown that $\max_H \text{err}(Q, I_0, \gamma^H, C^Q) = 0$, with $\max_H$ being taken over all dynamic networks H with at most n nodes. Now consider any other input vector I. Since P is input-stable, Theorem 11 immediately tells us that C^Q must be in both $A_1(I)$ and $A_2(I)$ as well. Hence $C^Q \in A_1(I) \cap A_2(I)$. Again by our proof above, we know that $\max_H \text{err}(Q, I, \gamma^H, C^Q) = 0$, with $\max_H$ being taken over all dynamic networks H with at most n nodes. This in turn means that $\max_I \max_H \text{err}(Q, I, \gamma^H, C^Q) = 0$. □