DEX: Self-Healing Expanders

Pandurangan, G., Robinson, P., & Trehan, A. (2014). DEX: Self-Healing Expanders. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International* (pp. 702-711) https://doi.org/10.1109/IPDPS.2014.78

Published in:
Parallel and Distributed Processing Symposium, 2014 IEEE 28th International

Document Version:
Peer reviewed version

DEX: Self-healing Expanders

Gopal Pandurangan
Div. of Mathematical Sciences
Nanyang Technological University
Singapore 637371
gopalpandurangan@gmail.com

Peter Robinson
Div. of Mathematical Sciences
Nanyang Technological University
Singapore 637371
peter.robinson@ntu.edu.sg

Amitabh Trehan
School of Electronics, Electrical Engineering and Computer Sciences
Queen's University Belfast
Belfast, NI, United Kingdom
a.trehan@qub.ac.uk

***Abstract*—We present a fully-distributed self-healing algorithm DEX that maintains a constant degree expander network in a dynamic setting. To the best of our knowledge, our algorithm provides the first efficient distributed construction of expanders — whose expansion properties hold *deterministically* — that works even under an all-powerful adaptive adversary that controls the dynamic changes to the network (the adversary has unlimited computational power and knowledge of the entire network state, can decide which nodes join and leave and at what time, and knows the past random choices made by the algorithm). Previous distributed expander constructions typically provide only *probabilistic* guarantees on the network expansion which *rapidly degrade* in a dynamic setting; in particular, the expansion properties can degrade even more rapidly under *adversarial* insertions and deletions.**

Our algorithm provides efficient maintenance and incurs a low overhead per insertion/deletion by an adaptive adversary: only $O(\log n)$ rounds and $O(\log n)$ messages are needed with high probability (n is the number of nodes currently in the network). The algorithm requires only a constant number of topology changes. Moreover, our algorithm allows for an efficient implementation and maintenance of a distributed hash table (DHT) on top of DEX with only a constant additional overhead.

Our results are a step towards implementing efficient self-healing networks that have *guaranteed* properties (constant bounded degree and expansion) despite dynamic changes.

I. Introduction

Modern networks (peer-to-peer, mobile, ad-hoc, Internet, social, etc.) are dynamic and increasingly resemble self-governed living entities with largely distributed control and coordination. In such a scenario, the network topology governs much of the functionality of the network. In what topology should such nodes (having limited resources and bandwidth) connect so that the network has effective communication channels with low latency for all messages, has constant degree, is robust to a limited number of failures, and nodes can quickly sample a random node in the network (enabling many randomized protocols)? The well known answer is that they should connect as a (constant degree) *expander* (see e.g., [1]). How should such a topology be constructed in a distributed fashion? The problem is especially challenging in a *dynamic* network, i.e., a network exhibiting churn with nodes and edges entering and leaving the system. Indeed, it is a fundamental problem to scalably build dynamic topologies that have the desirable properties of an expander graph (constant degree and expansion, regardless of the network size) in a distributed manner such that the expander properties are *always* maintained despite continuous network changes. Hence it is of both theoretical and practical interest to maintain expanders dynamically in an efficient manner.

Many previous works (e.g., [2], [3], [4]) have addressed the above problem, especially in the context of building dynamic P2P (peer-to-peer) networks. However, all these constructions provide only *probabilistic* guarantees of the expansion properties that *degrade rapidly* over a series of network changes (insertions and/or deletions of nodes/edges) — in the sense that expansion properties cannot be maintained ad infinitum due to their probabilistic nature[2] which can be a major drawback in a dynamic setting. In fact, the expansion properties can degrade even more rapidly under adversarial insertions and deletions (e.g., as in [3]). Hence, in a dynamic setting, guaranteed expander constructions are needed. Furthermore, it is important that the network maintains its expander properties (such as high conductance, robustness to failures, and fault-tolerant multi-path routing) *efficiently* even under dynamic network changes. This will be useful in efficiently building good overlay and P2P network topologies with expansion guarantees that do not degrade with time, unlike the above approaches.

Self-healing is a *responsive* approach to fault-tolerance, in the sense that it responds to an attack (or component failure) by changing the topology of the network. This approach works irrespective of the initial state of the network, and is thus orthogonal and complementary to traditional non-responsive techniques. Self-healing assumes the network

[1]Work supported in part by Nanyang Technological University grant M58110000, Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 2 grant MOE2010-T2-2-082, MOE AcRF Tier 1 grant MOE2012-T1-001-094, and by a grant from the United States-Israel Binational Science Foundation (BSF).

G. Pandurangan is also affiliated with Brown University, Providence, RI 02912, USA.

[2]For example, even if the network is guaranteed to be an expander with high probability (w.h.p.), i.e. a probability of $1 - 1/n^c$, for some constant c, in every step (e.g., as in the protocols of [3] and [2]), the probability of violating the expansion bound tends to 1 after some polynomial number of steps.

to be *reconfigurable* (e.g. P2P, wireless mesh, and ad-hoc networks), in the sense that changes to the topology of the network can be made on the fly. Our goal is to design an efficient distributed self-healing algorithm that maintains an expander despite attacks from an adversary.

Our Model: We use the self-healing model which is similar to the model introduced in [5], [6] and is briefly described here (the detailed model is described in Sec. II). We assume an adversary that repeatedly attacks the network. This adversary is adaptive and knows the network topology and our algorithm (and also previous insertions/deletions and all previous random choices), and it has the ability to delete arbitrary nodes from the network or insert a new node in the system which it can connect to any subset of nodes currently in the system. We also assume that the adversary can only delete or insert a single node at a time step. (Our algorithm can be extended to handle multiple insertions/deletions — cf. full paper [7]). The neighbors of the deleted or inserted node are aware of the attack in the same time step and the self-healing algorithm responds by adding or dropping edges (i.e. connections) between nodes. The computation of the algorithm proceeds in synchronous rounds and we assume that the adversary does not perform any more changes until the algorithm has finished its response. As typical in self-healing (see e.g. [5], [8], [6]), we assume that no other insertion/deletion takes place during the repair phase [3] (though our algorithm can be potentially extended to handle such a scenario). The goal is to minimize the number of distributed rounds taken by the self-healing algorithm to heal the network.

Our Contributions: In this paper, we present DEX, in our knowledge the first *distributed* algorithm to efficiently construct and dynamically maintain a constant degree expander network (under both insertions and deletions) under an all-powerful adaptive adversary. Unlike previous constructions (e.g.,[2], [3], [4], [9], [10]), the expansion properties always hold, i.e., the algorithm guarantees that the dynamic network *always* has a constant spectral gap (for some fixed absolute constant) despite continuous network changes, and has constant degree, and hence is a (sparse) expander[4]. The maintenance overhead of DEX is very low. It uses only local information and small-sized messages, and hence is scalable. The following theorem states our main result:

Theorem 1: Consider an adaptive adversary that observes the entire state of the network including all past random choices and inserts or removes a single node in every step. Algorithm DEX maintains a constant degree expander network that has a constant spectral gap. The algorithm takes $O(\log n)$ rounds and messages in the worst case (with high probability[5]) per insertion/deletion where n is the current network size. Furthermore, DEX requires only a constant number of topology changes.

Note that the above bounds hold w.h.p. for *every* insertion/deletion (i.e., in a worst case sense) and not just in an amortized sense. Our algorithm can be extended to handle multiple insertions/deletions per step (cf. full paper [7]). We also describe (cf. Sec. IV-B) how to implement a distributed hashtable (DHT) on top of our algorithm DEX, which provides insertion and lookup operations using $O(\log n)$ messages and rounds.

Our results answer some open questions raised in prior work. In [4], the authors ask: Can one can design a fully decentralized construction of dynamic expander topologies with *constant* overhead? The expander maintenance algorithms of [4] and [3] handle deletions much less effectively than additions; [4] also raises the question of handling deletions as effectively as insertions. Our algorithm handles even *adversarial* deletions as effectively as insertions.

Technical Contributions: Our approach differs from previous approaches to expander maintenance (e.g., [3], [2], [4]). Our approach *simulates* a virtual network (cf. Sec. III-A) on the actual (real) network. At a high level, DEX works by stepping between instances of the guaranteed expander networks (of different sizes as required) in the virtual graph. It maintains a *balanced mapping* (cf. Def. 2) between the two networks with the guarantee that the spectral properties and degrees of both are similar. The virtual network is maintained as a p-cycle expander (cf. Def. 1). Since the adversary is fully adaptive with complete knowledge of topology and past random choices, it is non-trivial to efficiently maintain *both* constant degree and constant spectral gap of the virtual graph. Our maintenance algorithm DEX uses randomization to defeat the adversary and exploits various key algorithmic properties of expanders, in particular, Chernoff-like concentration bounds for random walks ([11]), fast (almost) uniform sampling, efficient permutation routing ([12]), and the relationship between edge expansion and spectral gap as stated by the Cheeger Inequality (cf. Theorem 2.6 in [13]). Moreover, we use certain structural properties of the p-cycle and staggering of "complex" steps that require more involved recovery operations over multiple "simple" steps to achieve worst case $O(\log n)$ complexity bounds. It is technically and conceptually much more convenient to work on the (regular) virtual network and this can be a useful algorithmic paradigm in handling other dynamic problems as well.

Related Work and Comparison: Expanders are a very important class of graphs that have applications in various areas of computer science (e.g., see [13] for a survey) e.g. in distributed networks, expanders are used for solv-

[3]One way to think about this assumption is that insertion/deletion steps happen somewhat at a slower time scale compared to the time taken by the self-healing algorithm to repair; hence this motivates the need to design fast self-healing algorithms.

[4]The full paper [7] contains the formal definition of an expander and related concepts such as expansion, spectral gap, and Cheeger inequality.

[5]With high probability (w.h.p.) means with probability $\geqslant 1 - n^{-1}$.

ing distributed agreement problems efficiently[14], [15]. In distributed dynamic networks (cf. [15]) it is particularly important that the expansion does not degrade over time. There are many well known (centralized) expander construction techniques see e.g., [13]) .

As stated earlier, there are a few other works addressing the problem of distributed expander construction; however all of these are randomized and the expansion properties hold with probabilistic guarantees only. Figure III compares our algorithm with some known distributed expander construction algorithms. [3] give a construction where an expander is constructed by composing a small number of random Hamiltonian cycles. The probabilistic guarantees provided degrade rapidly, especially under adversarial deletions. [4] builds on the algorithm of [3] and makes use of random walks to add new peers with only constant overhead. However, it is not a fully decentralized algorithm. Both these algorithms handle insertions much better than deletions. Spanders [16] is a self-stabilizing construction of an expander network that is a spanner of the graph. [17] shows a way of constructing random regular graphs (which are good expanders, w.h.p.) by performing a series of random 'flip' operations on the graph's edges. [18] maintains an almost d-regular graph, i.e. with degrees varying around d, using uniform sampling to select, for each node, a set of expander-neighbors. The protocol of [2] gives a distributed algorithm for maintaining a sparse random graph under a stochastic model of insertions and deletions. [19] gives a dynamic overlay construction that is empirically shown to resemble a random k-regular graph and hence is a good expander. [20] gives a gossip-based membership protocol for maintaining an overlay in a dynamic network that under certain circumstances provides an expander.

In a model similar to ours, [21] maintains a DHT (Distributed Hash Table) in the setting where an adaptive adversary can add/remove $O(\log n)$ peers per step. Another paper which considers node joins/leaves is [10] which constructs a SKIP+ graph within $O(\log^2 n)$ rounds starting from any graph whp. Then, they also show that after an insert/delete operation the system recovers within $O(\log n)$ steps (like ours, which also needs $O(\log n)$ steps whp) and with $O(\log^4 n)$ messages (while ours takes $O(\log n)$ messages whp). However, the SKIP+ graph has an additional advantage that it is *self-stabilizing*, i.e., can recover from any initial state (as long as it is weakly connected). [10] assume (as do we) that the adversary rests while the network converges to a SKIP+ graph. It was shown in [9] that skip graphs contain expanders as subgraphs w.h.p., which can be used as a randomized expander construction. Skip graphs (and its variant SKIP+ [10]) are probabilistic structures (i.e., their expansion holds only with high probability) and furthermore, they are not of constant degree, their degree grows logarithmic in the network size. The work of [22] has guaranteed expansion (like ours). However, as pointed out in [9], its main drawback (unlike ours) is that their algorithm has a rather large overhead in maintaining the network.

A variety of self-healing algorithms deal with maintaining topological invariants on arbitrary graphs [5], [8], [6], [23], [24]. The self-healing algorithm *Xheal* of [8] maintains spectral properties of the network (while allowing only a small increase in stretch and degree), but it relied on a randomized expander construction and hence the spectral properties degraded rapidly. Using our algorithm as a subroutine, *Xheal* can be efficiently implemented with guaranteed spectral properties.

II. The Self-Healing Model

The model we are using is similar to the models used in [5], [8]. We now describe the details. Let $G = G_0$ be a small arbitrary graph[5] where nodes represent processors in a distributed network and edges represent the links between them. Each *step* $t \geqslant 1$ is triggered by a deletion or insertion of a single[6] node from G_{t-1} by the adversary, yielding an *intermediate network graph* U_t. The neighbors of the (inserted or deleted) node in the network U_t react to this change by adding or removing edges in U_t, yielding G_t — this is called *recovery or repair*. The distributed computation during recovery is structured into synchronous rounds. We assume that the adversary rests until the recovery is complete, and subsequently triggers the next step by inserting/deleting a node. During recovery, nodes can communicate with their neighbors by sending messages of size $O(\log n)$, which are neither lost nor corrupted. We assume that local computation (within a node) is free, which is a standard assumption in distributed computing (e.g. [25]). Our focus is only on the cost of communication (time and messages).

Initially, a newly inserted node v only knows its unique id (chosen by the adversary) and does not have any a priori knowledge of its neighbors or the current network topology. In particular, this means that a node u can only add an edge to a node w if it knows the id of w.

In case of an insertion, we assume that the newly added node is initially connected to a constant number of other nodes. This is merely a simplification; nodes are not malicious but faithfully follow the algorithm, thus we could explicitly require our algorithm to immediately drop all but a constant number of edges. The adversary is *fully adaptive* and is aware of our algorithm, the complete state of the current network including all past random choices. As typically the case (see e.g. [5], [8]), we assume that no other node is deleted or inserted until the current step has concluded (though our algorithm can be modified to handle such a scenario).

[6]See the full paper [7] for multiple insertions/deletions per step.

Algorithms	Expansion Guarantees	Adversary	Max Degree	Recovery Time	Messages	Topology Changes
Law-Siu[3][$]	Prob$\geqslant 1-1/n_0$	Oblivious	$O(d)$	$O(\log_d n)$	$O(d \log_d n)$	$O(d)$
Skip Graphs [9][‡]	w.h.p.[†]	Adaptive	$O(\log n)$	$O(\log^2 n)$	$O(\log^2 n)$	$O(\log n)$
Skip+ [10][!]	w.h.p.[†]	Adaptive	$O(\log n)$	$O(\log n)^\dagger$	$O(\log^4 n)$	$O(\log^4 n)^\dagger$
DEX (This paper)	Deterministic	Adaptive	$O(1)$	$O(\log n)^\dagger$	$O(\log n)^\dagger$	$O(1)$

[†] With high probability.
[$] n_0 is the initial network size. Parameter d = # of Hamiltonian cycles in 'healing' graph ($\mathbb{H}$).
[‡] Costs given under certain assumptions about key length.
[!] Skip+ is a self-stabilizing structure but costs here are for single join/leave operations once a valid skip+ graph is achieved.

Figure 1. Comparison of distributed expander constructions.

III. Preliminaries and Overview of Algorithm DEX

It is instructive to first consider the following natural (but inefficient) algorithms:

Flooding: First, we consider a naive flooding-based algorithm that also achieves guaranteed expansion and node degree bounds, albeit at a much larger cost: Whenever a node is inserted (or deleted), a neighboring node floods a notification throughout the entire network and every node, having complete knowledge of the current network graph, locally recomputes the new expander topology. While this achieves a logarithmic runtime bound, it comes at the cost of using $\Theta(n)$ messages in *every* step and, in addition, might also result in $O(n)$ topology changes, whereas our algorithms requires only polylogarithmic number of messages and constant topology changes on average.

Maintaining Global Knowledge: As a second example of a straightforward but inefficient solution, consider the algorithm that maintains a global knowledge at some node p, which keeps track of the entire network topology. Thus, every time some node u is inserted or deleted, the neighbors of u inform p of this change, and p then proceeds to update the current graph using its global knowledge. However, when p itself is deleted, we would need to transfer all of its knowledge to a neighboring node q, which then takes over p's role. This, however, requires at least $\Omega(n)$ rounds, since the *entire* knowledge of the network topology needs to be transmitted to q.

Our Approach — Algorithm DEX: As mentioned in Sec. II, the actual (real) network is represented by a graph where nodes correspond to processors and edges to connections. Our algorithm maintains a second graph, which we call the *virtual graph* where the vertices do not directly correspond to the real network but each (virtual) vertex in this graph is simulated by a (real) node [7] in the network. The topology of the virtual graph determines the connections in the actual network. For example, suppose that node u simulates vertex z_1 and node v simulates vertex z_2. If there is an edge (z_1, z_2) according to the virtual graph, then our algorithm maintains an edge between u and v in the actual network. In other words, a real node may be simulating multiple virtual vertices and maintaining their edges according to the virtual graph.

Figure 2 on page 4 shows a real network (on the right) whose nodes (shaded rectangles) simulate the virtual vertices of the virtual graph (on the left).

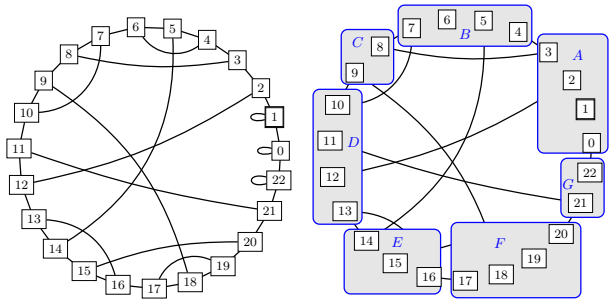

Figure 2. A 4-balanced virtual mapping of a p-cycle expander to the network graph. On the left is a (virtual) 3-regular 23-cycle expander on $\mathbb{Z}_{23}$; on the right is the network G_t with (real) nodes $\{A, \ldots, G\}$.

In our algorithm, we maintain this virtual graph and show that preserving certain desired properties (in particular, constant expansion and degree) in the virtual graph leads to these properties being preserved in the real network. Our algorithm achieves this by maintaining a "balanced load mapping" (cf. Def. 3) between the virtual vertices and the real nodes as the network size changes at the will of the adversary. The balanced load mapping keeps the number of virtual nodes simulated by any real node to be a constant — this is crucial in maintaining the constant degree bound. We next formalize the notions of virtual graphs and balanced mappings.

A. Virtual Graphs and Balanced Mappings

Consider some graph G and let λ_G denote the *second largest eigenvalue* of the adjacency matrix of G. The *contraction* of vertices z_1 and z_2 produces a graph H where z_1 are z_2 merged into a single vertex z that is adjacent to all vertices to which z_1 or z_2 were adjacent in G. We extensively make use of the fact that this operation leaves the *spectral gap* $1 - \lambda_G$ intact, cf. Lemma 1.15 in [26].

As mentioned earlier, our virtual graph consists of virtual vertices simulated by real nodes. Intuitively speaking, we

[7] Henceforth, we reserve the term "vertex" for vertices in a virtual graph and (real) "node" for vertices in the real network.

can think of a real node simulating z_1 and z_2 as a vertex contraction of z_1 and z_2. The above stated contraction property motivates us to use an expander family as virtual graphs. We now define the p-cycle expander family, which we use as virtual graphs in this paper. Essentially, we can think of a p-cycle as a numbered cycle with some chord-edges between numbers that are multiplicative inverses of each other. It was shown in [27] that this yields an infinite family of 3-regular expander graphs with a constant eigenvalue gap. Figure 2 shows a 23-cycle.

*Definition 1 (**p**-cycle, cf. [13]):* For any prime number p, we define the following graph $\mathcal{Z}(p)$. The vertex set of $\mathcal{Z}(p)$ is the set $\mathbb{Z}_p = \{0, \ldots, p-1\}$ and there is an edge between vertices x and y if and only if one of the following conditions hold: (1) $y = (x+1) \mod p$, (2) $y = (x-1) \mod p$, or (3) if $x, y > 0$ and $y = x^{-1}$. Moreover, vertex 0 has a self-loop.

At any point in time t, our algorithm maintains a mapping from the virtual vertices of a p-cycle to the actual network nodes. We use the notation $\mathcal{Z}_t(p)$ when $\mathcal{Z}(p)$ is the p-cycle that we are using for our mapping in step t. (We omit p and simply write $\mathcal{Z}_t$ if p is irrelevant or clear from the context.) At any time t, each real node simulates at least one virtual vertex (i.e. a vertex in the p-cycle) and all its incident edges as required by Def. 1, i.e., the real network can be considered a contraction of the virtual graph; see Figure 2 on page 4 for an example. Formally, this defines a function that we call a virtual mapping:

*Definition 2 (***Virtual mapping):** For step $t \geqslant 1$, consider a surjective map $\Phi_t : V(\mathcal{Z}_t) \to V(G_t)$ that maps every virtual vertex of the virtual graph $\mathcal{Z}_t$ to some (real) node of the network graph G_t. Suppose that there is an edge $(\Phi_t(z_1), \Phi_t(z_2)) \in E(G_t)$ if and only if there is an edge $(z_1, z_2) \in E(Z_t)$, for all nodes $z_1, z_2 \in V(Z_t)$. Then we call Φ_t a *virtual mapping*. Moreover, we say that node $u \in V(G_t)$ is a real node that *simulates* virtual vertices $z_1, \cdots, z_k$, if $u = \Phi_t(z_1) = \cdots = \Phi_t(z_k)$.

In the standard metric spaces on $\mathcal{Z}_t$ and G_t induced by the shortest-path metric Φ is a surjective metric map since distances do not increase:

Fact 1: Let $\mathrm{dist}_H(u, v)$ denote the length of the shortest path between u and v in graph H. Any virtual mapping Φ_t guarantees that $\mathrm{dist}_{Z_t}(z_1, z_2) \geqslant \mathrm{dist}_{G_t}(\Phi(z_1), \Phi(z_2))$, for all $z_1, z_2 \in \mathcal{Z}_t$.

We simply write Φ instead of Φ_t when t is irrelevant.

We consider the vertices of $\mathcal{Z}_t$ to be partitioned into disjoint sets of vertices that we call *clouds* and denote the cloud to which a vertex z belongs as $\textsc{cloud}(z)$. Whereas initially we can think of a cloud as the set of virtual vertices simulated at some node in G_t, this is not true in general due to load balancing issues, as we discuss in Section IV. We are only interested in virtual mappings where the *maximum cloud size* is bounded by some universal constant ζ, which is crucial for maintaining a constant node degree. For our p-cycle construction, it holds that $\zeta \leqslant 8$.

We now formalize the intuition that the expansion of the virtual p-cycle carries over to the network, i.e., the second largest eigenvalue λ_{G_t} of the real network is bounded by λ_{Z_t} of the virtual graph. Observe that we can obtain G_t from $\mathcal{Z}_t$ by contracting vertices. That is, we contract vertices z_1 and z_2 if $\Phi(z_1) = \Phi(z_2)$. According to Lemma 1.15 in [26] these operations do not increase λ_{G_t} and thus we have shown the following:

Lemma 1: Let $\Phi_t : \mathcal{Z}_t \to G_t$ be a virtual mapping. Then it holds that $\lambda_{G_t} \leqslant \lambda_{\mathcal{Z}_t}$.

Next we formalize the notion that our real nodes simulate at most a constant number of nodes. Let $\textsc{Sim}_t(u) = \Phi_t^{-1}(u)$ and define the *load of a node u in graph G_t* as the number of vertices simulated at u, i.e., $\textsc{Load}_t(u) = |\textsc{Sim}_t(u)|$. Note that due to locality, node u does not necessarily know the mapping of other nodes.

*Definition 3 (***Balanced mapping):** Consider a step t. If there exists a constant C s.t. $\forall u \in G_t \colon \textsc{Load}_t(u) \leqslant C$, then we say that Φ_t is a *C-balanced virtual mapping* and say that *G_t is C-balanced*.

Figure 2 on page 4 shows a balanced virtual mapping. At any step t, the degree of a node $u \in G_t$ is exactly $3.\textsc{Load}_t(u)$ since we are using the 3-regular p-cycle as a virtual graph. Thus our algorithm strives to maintain a constant bound on $\textsc{Load}_t(u)$, for all t. Given a virtual mapping Φ_t, we define the sets

$$\textsc{Low}_t = \{u \in G_t \colon \textsc{Load}_t(u) \leqslant 2\zeta\}; \quad (1)$$

$$\textsc{Spare}_t = \{u \in G_t \colon \textsc{Load}_t(u) \geqslant 2\}. \quad (2)$$

Intuitively speaking, $\textsc{Low}_t$ contains nodes that do not simulate too many virtual vertices, i.e., have relatively low degree, whereas $\textsc{Spare}_t$ is the set of nodes that simulate at least 2 vertices each. When the adversary deletes some node u, we need to find a node in $\textsc{Low}_t$ that takes over the load of u. Upon a node v being inserted, on the other hand, we need to find a node in $\textsc{Spare}_t$ that can spare a virtual vertex for v, while maintaining the surjective property of the virtual mapping.

IV. Expander Maintenance Algorithm

We describe our maintenance algorithm DEX (the complete pseudo-code is in the full paper [7]) and prove (omitted proofs are included in the full paper [7]) the performance claims of Theorem 1. We start with a small initial network G_0 of some appropriate constant and assume there is a virtual mapping from a p-cycle $\mathcal{Z}_0(p_0)$ where p_0 is the smallest prime number in the range $(4n_0, 8n_0)$. The existence of p_0 is guaranteed by Bertrand's postulate [28]. (Since G_0 is of constant size, nodes can compute the current network size n_0 and $\mathcal{Z}_0(p_0)$ in a constant number of rounds in a centralized manner.) Starting out from this initial expander,

we seek to guarantee expansion ad infinitum, for any number of adversarial insertions and deletions.

We always maintain the invariant that each real node simulates at least one (i.e. the virtual mapping is surjective) and at most a constant number of virtual p-cycle vertices. The adversary can either insert or delete a node in every step. In either case, our algorithm reacts by doing an appropriate redistribution of the virtual vertices to the real nodes with the goal of maintaining a C-balanced mapping (cf. Definition 3).

Depending on the operations employed by the algorithm, we classify the response of the algorithm for a given step t as being either a *type-1 recovery* or a *type-2 recovery* and call t a *type-1 recovery step* (resp. type-2 recovery step). Type-1 recovery is very efficient, as (w.h.p.) it suffices to execute a single random walk of $O(\log n)$ length.

It is somewhat more complicated to show a worst case $O(\log n)$ performance for type-2 recovery: Here, the current virtual graph is either *inflated* or *deflated* to ensure a C-balanced mapping (i.e. bounded degrees). For the sake of exposition, we first present a simpler way to handle inflation and deflation, which yields amortized complexity bounds. We then describe a more complicated algorithm for type-2 recovery that yields the claimed *worst case* complexity bounds of $O(\log n)$ rounds and messages, and $O(1)$ topology changes per step with high probability. The first (simplified) approach (cf. Sec. IV-B) replaces the entire virtual graph by a new virtual graph of appropriate size in a single step. This requires $O(n)$ topology changes and $O(n \log^2 n)$ message complexity, because all nodes complete the inflation/deflation in one step. Since there are at least $\Omega(n)$ steps with type-1 recovery between any two steps where inflation or deflation is necessary, we can nevertheless amortize their cost and get the amortized performance bounds of $O(\log n)$ rounds and $O(\log^2 n)$ messages (cf. Cor. 1 in the full paper [7]). We then present an improved (but significantly more complex) way of handling inflation (resp. deflation), by *staggering* these inflation/deflation operations across the recovery of the next $\Theta(n)$ following steps while retaining constant expansion and node degrees. This yields a $O(\log n)$ *worst case bounds for both messages and rounds* for *all* steps as claimed by Theorem 1. In terms of expansion, the (amortized) inflation/deflation approach yields a spectral gap no smaller than of the p-cycle, the improved worst case bounds of the 2nd approach come at the price of a slightly reduced, but still constant, spectral gap. Algorithm IV.1 presents a high-level pseudo code description of our approach.

Case 1: Adversary inserts a node u:
Try to find a spare vertex for u via a random walk (**type-1 recovery**).
if type-1 recovery fails **then**
  if most nodes simulate only 1 vertex **then**
    Perform **type-2 recovery** by inflating.
  else
    Retry type-1 recovery until it succeeds.

Case 2: Adversary deletes a node u:
Try distributing vertices that were simulated at u via random walks (**type-1 recovery**).
if type-1 recovery fails **then**
  if most nodes simulate many vertices **then**
    Perform **type-2 recovery** by deflating.
  else
    Retry type-1 recovery until it succeeds.

Algorithm IV.1: High-level overview of our algorithm DEX

A. Type-1 Recovery

When a node u is inserted, a neighboring node v initiates a random walk of length at most $\Theta(\log n)$ to find a "spare" virtual vertex, i.e., a virtual vertex z that is simulated by a node $w \in \text{SPARE}_{G_{t-1}}$. Assigning this virtual vertex z to the new node u, ensures a surjective mapping of virtual vertices to real nodes at the end of the step.

When a node u is deleted, on the other hand, the notified neighboring node v also initiates random walks, except this time with the aim of redistributing the deleted node u's virtual vertices to the remaining real nodes in the system. We assume that every node v has knowledge of $\text{LOAD}_{G_{t-1}}(w)$, for each of its neighbors u. (This can be implemented with constant overhead, by simply updating neighboring nodes when the respective $\text{LOAD}_{G_{t-1}}$ changes.) Since the deleted node u might have simulated multiple vertices, node v initiates a random walk for each $z \in \text{LOAD}_{G_{t-1}}(u)$, to find a node $w \in \text{LOW}_{G_{t-1}}$ to take over virtual vertex z. In a nutshell, type-1 recovery consists of (re)balancing the load of virtual vertices to real nodes by performing random walks. Rebalancing the load of a deleted node succeeds with high probability, as long as θn nodes are in $\text{LOW}_{G_{t-1}}$, where the *rebuilding parameter* θ is a fixed constant. For our analysis, we require that $\theta \leqslant 1/(68\zeta + 1)$, where $\zeta \leqslant 8$ is the maximum (constant) cloud size given by the p-cycle construction. Analogously, for insertion steps, finding a spare vertex will succeed w.h.p. if $\text{SPARE}_{G_{t-1}}$ has size $\geqslant \theta n$. If the size is below θn, we handle the insertion (resp. deletion) by performing an inflation (resp. deflation) as explained below. Thus we formally define a step t to be a *type-1 step*, if either (1) a node is inserted in t and $|\text{SPARE}_{G_{t-1}}| \geqslant \theta n$ or (2) a node is deleted in t and $|\text{LOW}_{G_{t-1}}| \geqslant \theta n$.

If a random walk fails to find an appropriate node, we do not directly start an inflation resp. deflation, but first deterministically count the network size and sizes of $\text{SPARE}_{G_{t-1}}$ and $\text{LOW}_{G_{t-1}}$ by simple aggregate flooding (cf. Procedures `computeLow` and `computeSpare`). We repeat the random walks, if it turns out that the respective set indeed comprises $\geqslant \theta n$ nodes. As we will see below, this allows us to deterministically guarantee constant node degrees. We will only consider the case where a node is deleted and defer the (analogously handled) insertions to the full paper [7].

Lemma 2: Consider a step t and suppose that Φ_{t-1} is a 4ζ-balanced virtual map. There exists a constant ℓ such that the following hold w.h.p: If $|\text{LOW}_{G_{t-1}}| \geqslant \theta n$ and some node u is deleted, then, for each of the (at most $4\zeta \in O(1)$) vertices simulated at u, the initiated random walk reaches a node in $\text{SPARE}_{G_{t-1}}$ in $\ell \log n$ rounds. That is, w.h.p. type-1 recovery succeeds in $O(\log n)$ messages and rounds, and a constant number of edges are changed.

Proof: The main idea of the proof is to instantiate a concentration bound for random walks on expander graphs [11]. By assumption, the mapping of virtual vertices to real nodes is 4ζ-balanced before the deletion occurs. Thus we only need to redistribute a constant number of virtual vertices when a node is deleted.

We now present the detailed argument. By assumption we have that $|\text{LOW}| = an \geqslant \theta n$, for a constant $0 < a < 1$. We start a random walk of length $\ell \log n$ for some appropriately chosen constant ℓ (determined below). We need to show that (w.h.p.) the walk hits a node in LOW. According to the description of type-1 recovery for handling deletions, we perform the random walk on the graph G'_t, which modifies $G_{t-1} \setminus \{u\}$, by transferring all virtual vertices (and edges) of the deleted node u to the neighbor v. Thus, for the second largest eigenvalue $\lambda = \lambda_{G'_t}$, we know by Lemma 1 that $\lambda \leqslant \lambda_{G_{t-1}}$. Consider the normalized $n \times n$ adjacency matrix M of G'_t. It is well known (e.g., Theorem 7.13 in [29]) that a vector π corresponding to the stationary distribution of a random walk on G_{t-1} has entries $\pi(x) = \frac{d_x}{2|E(G'_t)|}$ where d_x is the degree of node x. By assumption, the network G_{t-1} is the image of a 4ζ-balanced virtual map. This means that the maximum degree Δ of any node in the network is $\Delta \leqslant 12\zeta$, and since the p-cycle is a 3-regular expander, every node has degree at least 3. If the adversary deletes some node in step t, the maximum degree of one of its neighbors can increase by at most Δ. Therefore, the maximum degree in U_t and thus G'_t is bounded by 2Δ, which gives us the bound

$$\pi(x) \geqslant 3/(2\Delta n), \tag{3}$$

for any node $x \in G'_t$. Let ρ be the actual number of nodes in LOW that the random walk of length $\ell \log n$ hits. We define $\mathbf{q}$ to be an n-dimensional vector that is 0 everywhere except at the index of u in M where it is 1. Let $\mathcal{E}$ be the event that $\ell \log n \cdot \pi(\text{LOW}) - \rho \geqslant \gamma$, for a fixed $\gamma \geqslant 0$. That is, $\mathcal{E}$ occurs if the number of nodes in LOW visited by the random walk is far away ($\geqslant \gamma$) from its expectation.

In the remainder of the proof, we show that $\mathcal{E}$ occurs with very small probability. Applying the concentration bound of [11] yields that $\Pr[\mathcal{E}] \leqslant \left(1 + \frac{\gamma(1-\lambda)}{10\ell \log n}\right) \cdot \left\| \frac{\mathbf{q}}{\sqrt{\pi}} \right\|_2 \cdot e^{\frac{-\gamma^2(1-\lambda)}{20\ell \log n}}$, where $\mathbf{q}/\sqrt{\pi}$ is a vector with entries $(\mathbf{q}/\sqrt{\pi})(x) = \mathbf{q}(x)/\sqrt{\pi(x)}$, for $1 \leqslant x \leqslant n$. By (3), we know that $\pi(\text{LOW}) \geqslant 3a/2\Delta$. To guarantee that we find a node in LOW w.h.p. even when $\pi(\text{LOW})$ is small, we must set $\gamma = \frac{3a\ell}{2\Delta} \log n$. Moreover, (3) also gives us the bound $\|\mathbf{q}/\sqrt{\pi}\|_2 \leqslant \sqrt{2\Delta/3}\sqrt{n}$. We define $C = \left(1 + \frac{3a}{20\Delta}\right)\sqrt{2\Delta/3}$. Plugging these bounds into the bounds on $\Pr[\mathcal{E}]$, shows that $\Pr[\mathcal{E}] \leqslant C\sqrt{n}e^{\left(-\frac{(3a\ell/2\Delta)^2(1-\lambda)\log n}{20\ell}\right)} = Cn^{\left(\frac{1}{2} - \frac{9a^2\ell(1-\lambda)}{80\Delta^2}\right)}$. To ensure that event $\mathcal{E}$ happens with small probability, it is sufficient if the exponent of n is smaller than $-C$, which is true for sufficiently large ℓ. Since θ, Δ, and the spectral gap $1 - \lambda$ are all $O(1)$, it follows that ℓ is a constant too and thus the running time of one random walk is $O(\log n)$ with high probability. Recall that node v needs to perform a random walk for each of the virtual vertices that were previously simulated by the deleted node u; there are at most $4\zeta \in O(1)$ such vertices, since we assumed that Φ_{t-1} is 4ζ balanced. Therefore, all random walks take $O(\log n)$ rounds in total (w.h.p.).

Note that we only transfer a constant number of virtual vertices to a new nodes in type-1 recovery steps, i.e., the number of topology changes is constant. ■

Lemma 3 (Worst Case Bounds Type-1 Rec.): If type-1 recovery is performed in t and G_{t-1} is 4ζ-balanced, it holds that

(a) G_t is 4ζ-balanced,
(b) step t takes $O(\log n)$ (w.h.p.), rounds,
(c) nodes send $O(\log n)$ messages in step t (w.h.p.), and
(d) the number of topology changes in t is constant.

Assumption: the adversary attaches inserted node u to arbitrary node v.

// Try to perform a **type-1 recovery**:

1: Node v initiates a random walk of length $\ell \log n$ by generating a token τ and sending it to a neighbor u' chosen uniformly at random, but excluding u. Node u' in turn forwards τ by chosing a neighbor at random and so forth. Note that the newly inserted node u is excluded from being reached by the random walk. The walk terminates upon reaching a node $w \in$ SPARE (cf. Equation (2)).
2: **if** found node $w \in$ SPARE **then**
3: Transfer a virtual vertex and all its edges (according to the virtual graph) from w to u. Remove edge between u and v unless required by $\mathcal{Z}_t$.
4: **else** // the walk did not hit a node in SPARE; perform **type-2 recovery** if necessary:
5: Determine current network size n and |SPARE| via `computeSpare` (cf. Algorithm IV.4).
6: **if** $|\text{SPARE}| < \theta n$ **then** // Perform type-2 recovery:
7: Invoke type-2 recovery.
8: **else** // Sufficiently many nodes with spare virtual vertices are present but the walk did not find them. Happens with probability $\leqslant 1/n$.
9: Repeat from Line 1.

Algorithm IV.2: `insertion(`u, θ`)`

B. Type-2 Recovery: Inflating and Deflating

Recall that we perform type-1 recovery in step t, as long as θn nodes are in $\text{SPARE}_{G_{t-1}}$ when a node is inserted, resp. in $\text{LOW}_{G_{t-1}}$, upon a deletion.

Assumption: adversary deletes an arbitrary node u which simulated k virtual vertices. (We prove that $k \in O(1)$).

1: A (former) neighbor v of node u attaches all edges of u to itself.

// Try to perform a **type-1 recovery**:

2: **for** each of the k vertices **do**

3: Node v initiates a random walk of length $\ell \log n$ by generating a token τ and sending it to a uniformly at random chosen neighbor u'. Node u' in turn forwards τ by chosing a neighbor at random and so forth. The walk terminates upon reaching a node $w \in$ LOW (cf. Equation (1)).

4: **if** all random walks found nodes $w_1, \ldots, w_k \in$ LOW: **then**

5: Distribute the virtual vertices of u and their respective edges (according to the virtual graph) from v to $w_1, \ldots, w_k$.

6: **else** // Some of the random walks did not find a node in LOW; perform **type-2 recovery** if necessary:

7: Determine network size n and $|\text{LOW}|$ via `computeLow` (cf. Algorithm IV.4).

8: **if** $|\text{LOW}| < \theta n$ **then** // Perform type-2 recovery:

9: Invoke type-2 recovery.

10: **else** // Sufficiently many nodes with low load are present but the walk(s) did not find them. This happens with probability $\leqslant 1/n$:

11: Repeat from Line 3.

Algorithm IV.3: Procedure `deletion`(u, θ)

Given: DIAM is the diameter of $\mathcal{Z}_t$ (i.e. DIAM $\in O(\log n)$).

1: Node u broadcasts an aggregation request to all its neighbors. In addition to the network size, this request indicates whether to compute $|\text{LOW}|$ or $|\text{SPARE}|$. That is, the request of u traverses the network in a BFS-like manner and then returns the aggregated values to u.

2: If a node w receives this request from some neighbor, it computes the aggregated maximum value, according to whether $w \in$ SPARE for `computeSpare` (resp. $w \in$ LOW for `computeLow`).

3: If node w has received the request for the first time, w forwards it to all neighbors (except v).

4: Once the entire network has been explored this way, i.e., the request has been forwarded for DIAM rounds, the aggregated maximum values of the network size and $|\text{LOW}|$ (resp. $|\text{SPARE}|$) are sent back to u, which receives them after $\leqslant 2$DIAM rounds.

Algorithm IV.4: `computeSpare` and `computeLow`.

Inflating the Virtual Graph: If node v fails to find a spare node for a newly inserted neighbor and computes that $|\text{SPARE}_{G_{t-1}}| < \theta n$, i.e., only few nodes simulate multiple virtual vertices each, it invokes Procedure `simplifiedInfl` (cf. full paper [7]), which consists of two phases:

Phase 1: Constructing a Larger p-Cycle: Node v initiates replacing the current p-cycle $\mathcal{Z}_{t-1}(p_i)$ with the larger p-cycle $\mathcal{Z}_t(p_{i+1})$, for some prime number $p_{i+1} \in (4p_i, 8p_i)$. This rebuilding request is forwarded throughout the entire network to ensure that after this step, every node uses the exact same new p-cycle $\mathcal{Z}_t$. Intuitively speaking, each virtual vertex of $\mathcal{Z}_{t-1}$ is replaced by a cloud of (at most $\zeta \leqslant 8$) virtual vertices of $\mathcal{Z}_t$ and all edges are updated such that G_t is a virtual mapping of $\mathcal{Z}_t$.

For simplicity, we use x to denote both: an integer $x \in \mathbb{Z}_p$ and also the associated vertex in $V(\mathcal{Z}_t(p))$. At the beginning of step t, all nodes are in agreement on the current virtual graph $\mathcal{Z}_{t-1}(p_i)$, in particular, every node knows the prime number p_i. To get a larger p-cycle, all nodes deterministically compute the (same) smallest prime number $p_{i+1} \in (4p_i, 8p_i)$, i.e., $V(\mathcal{Z}_t(p_{i+1})) = \mathbb{Z}_{p_{i+1}}$. (Local computation happens instantaneously and does not incur any cost (cf. Sec. II).) Bertrand's postulate [28] states that for every $n > 1$, there is a prime between n and $2n$, which ensures that p_{i+1} exists. Every node u needs to determine the new set of vertices in $\mathcal{Z}_t(p_{i+1})$ that it is going to simulate: Let $\alpha = \frac{p_{i+1}}{p_i} \in O(1)$. For every currently simulated vertex $x \in \text{SIM}_{G_{t-1}}(u)$, node u computes the constant $c(x) = \lfloor \alpha(x+1) \rfloor - \lfloor \alpha x \rfloor - 1$, and replaces x with the new virtual vertices $y_0, \ldots, y_{c(x)}$ where $y_j = (\lfloor \alpha x \rfloor + j) \mod p_{i+1}$, for $0 \leqslant j \leqslant c(x)$. Note that the vertices $y_0, \ldots, y_{c(x)}$ form a cloud (cf. Sec. III-A) where the maximum cloud size is $\zeta \leqslant 8$. This ensures that the new virtual vertex set is a bijective mapping of $\mathbb{Z}_{p_{i+1}}$.

Next, we describe how we find the edges of $\mathcal{Z}_t(p_{i+1})$: First, we add new cycle edges (i.e. edges between x and $x+1 \mod p_{i+1}$), which can be done in constant time by using the cycle edges of the previous virtual graph $\mathcal{Z}_{t-1}(p_i)$. For every x that u simulates, we need to add an edge to the node that simulates vertex x^{-1}. Since this needs to be done by the respective simulating node of every virtual vertex, this corresponds to solving a permutation routing instance. Corollary 7.7.3 of [12] states that, for any bounded degree expander with n nodes, n packets, one per node, can be routed (even online) according to an arbitrary permutation in $O(\frac{\log n (\log \log n)^2}{\log \log \log n})$ rounds w.h.p. Note that every node in the network knows the exact topology of the current virtual graph (but not necessarily of the network graph G_t), and can hence calculate all routing paths in this graph, which map to paths in the actual network (cf. Fact 1). Since every node simulates a constant number of vertices, we can find the route to the respective inverse by solving a constant number of permutation routing instances.

Phase 2: Rebalancing the Load: Once the new virtual graph $\mathcal{Z}_t(p_{i+1})$ is in place, each real node simulates a greater number (by a factor of at most ζ) of virtual vertices and now a random walk is guaranteed to find a spare virtual vertex on the first attempt with high probability, according to Lemma 2.(a). At the beginning of the step, the virtual mapping Φ_{t-1} was 4ζ-balanced. This, however, is not necessarily the case after Phase 1, i.e., replacing $\mathcal{Z}_{t-1}$ by $\mathcal{Z}_t$. A node could have been simulating 4ζ virtual vertices *before* `simplifiedInfl` was invoked and now might be simulating $4\zeta^2$ vertices of $\mathcal{Z}_t(p_{i+1})$. In fact, this can be the case for a θ-fraction of the nodes. To ensure a 4ζ-balanced

mapping at the end of step t, we thus need to rebalance these additional vertices among the other (real) nodes. Note that this is always possible, since $(1-\theta)n$ nodes had a load of 1 before invoking `simplifiedInfl` and simulate only ζ virtual vertices each at the end of Phase 1. A node v that has a load of $k' > 4\zeta$ vertices of $\mathcal{Z}_t(p_{i+1})$, proceeds as follows, for each vertex z of the (at most constant) vertices that it needs to redistribute: Node v marks all of its vertices as *full* and initiates a random walk of length $\Theta(\log n)$ on the virtual graph $\mathcal{Z}_t(p_{i+1})$, which is simulated on the actual network. If the walk ends at a vertex z' simulated at some node w that is not marked as *full*, and no other random walk simultaneously ended up at z', then v transfers z to w. This ensures that z is now simulated at a node that had a load of $< 4\zeta$. A node w immediately marks all of its vertices as *full*, once its load reaches 2ζ. Node v repeatedly performs random walks until all of the $k' - 4\zeta$ vertices are transfered to other nodes.

Deflating the Virtual Graph: When the load of all but θn nodes exceeds 2ζ and some node u is deleted, the high probability bound of Lemma 2 for the random walk invoked by neighbor v no longer applies. In that case, node v invokes Procedure `simplifiedDefl` to reduce the overall load (cf. full paper [7]. Analogously as `simplifiedInfl`, Procedure `simplifiedDefl` consists of two phases:

Phase 1: Constructing a Smaller p-Cycle: To reduce the load of simulated vertices, we replace the current p-cycle $Z_{t-1}(p_i)$ with a smaller p-cycle $\mathcal{Z}_t(p_s)$ where p_s is a prime number in the range $(p_i/8, p_i/4)$.

Let $\alpha = p_i/p_s$. Any virtual vertex $x \in \mathcal{Z}_{t-1}(p_i)$, is (surjectively) mapped to some $y_x \in \mathcal{Z}_t(p_s)$ where $y = \lfloor x/\alpha \rfloor$. Note that we only add y to $V(\mathcal{Z}_t(p_s))$ if there is no smaller $x' \in \mathcal{Z}_{t-1}(p_i)$ that yields the same y. This mapping guarantees that, for any element in $\mathbb{Z}_{p_s}$, we have exactly 1 virtual vertex in $\mathcal{Z}_t(p_s)$: Suppose that there is some $y \in \mathbb{Z}_{p_s}$ that is not hit by our mapping, i.e., for all $x \in \mathbb{Z}_{p_i}$, we have $y > \lfloor \frac{x}{\alpha} \rfloor$. Let x' be the smallest integer such that $y = \lfloor \frac{x'}{\alpha} \rfloor$. For such an x', it must hold that $\alpha y \leqslant x' < \alpha(y+1)$. Since $\alpha > 1$, clearly x' exists. By assumption, we have $x' \geqslant p_i$, which yields $\lfloor p_i/\alpha \rfloor \leqslant \lfloor x'/\alpha \rfloor = y < p_s$. Since $p_s = p_i/\alpha$, we get $\lfloor p_s \rfloor < p_s$, which is a contradiction to $p_s \in \mathbb{N}$. Therefore, we have shown that $\mathbb{Z}_s \subseteq V(\mathcal{Z}_t(p_s))$. The opposite set inclusion can be shown similarly.

To add cycle edges and the edge between y and y^{-1}, we proceed as in Phase 1 of `simplifiedInfl`, i.e., we solve permutation routing on $\mathcal{Z}_{t-1}(p_i)$, taking $O(\frac{\log n (\log\log n)^2}{\log\log\log n})$ rounds.

Phase 2: Ensuring a Virtual Mapping: After Phase 1 is complete, the replacement of multiple virtual vertices in $\mathcal{Z}_{t-1}(p_i)$ by a single vertex in $\mathcal{Z}_t(p_s)$, might lead to the case where some nodes are no longer simulating *any* virtual vertices. A node that currently does not simulate a vertex, marks itself as *contending* and repeatedly keeps initiating random walks on $\mathcal{Z}_t(p_s)$ (that are simulated on the actual network graph) to find spare vertices.

Worst Case Bounds for Type-2 Recovery: Whereas Lemma 3 shows $O(\log n)$ worst case bounds for steps with type-1 recovery, handling of type-2 recovery that we have described so far yields *amortized* polylogarithmic performance guarantees on messages and rounds w.h.p. per step. We now present a more complex algorithm for type-2 recovery that yields worst case logarithmic bounds on messages and rounds per step (w.h.p.). The main idea of Procedures `inflate` and `deflate` (cf. full paper [7]) is to spread the type-2 recovery over $\Theta(n)$ steps of type-1 recovery, while still retaining constant node degrees and spectral expansion in *every* step. The details of these procedures are described in the full paper [7].

These operations are orchestrated by a *coordinator* node, which is the node that currently hosts the virtual vertex with integer-label 0; the coordinator keeps track of additional information (requiring $O(\log n)$ memory): the current network size n and the sizes of LOW and SPARE (but not the actual network topology), as follows:

Recall that we start out with an initial network of constant size, thus initially coordinator w can compute these values with constant overhead. If an insertion or deletion of some neighbor of v occurs and the algorithm performs type-1 recovery, then v informs coordinator w of the changes to the network size and the sizes of SPARE and LOW (by routing a message along a shortest path in $\mathcal{Z}_{t-1}(p_i)$) at the end of the type-1 recovery. Node v itself simulates some vertex $x \in \mathbb{Z}_{p_i}$ and hence can locally compute a shortest path from x to 0 (simulated at w) according to the edges in $\mathcal{Z}_t(p_i)$ (cf. Fact 1). The neighbors of w replicate w's state and update their copy in every step. If the coordinator w itself is deleted, the neighbors transfer its state to the new coordinator that subsequently simulates 0. The coordinator state requires only $O(\log n)$ bits and thus can be sent in 1 message. Keep in mind that the coordinator does *not* keep track of the actual network topology or SPARE and LOW, as this would require $\Omega(n)$ rounds for transferring the state to a new coordinator.

The following lemma summarizes the worst case bounds for staggered inflation/deflation via a coordinator node (cf. full paper [7] for the proof):

Lemma 4 (Worst Case Bounds Type-2 Rec.): Suppose that nodes initiate either `inflate` or `deflate` during recovery in a step t_0 and G_{t_0-1} is 4ζ-balanced. Then, for all $t \in [t_0, t_0+T]$ where $T = \lceil 2\theta n \rceil$ the following hold:

(a) Every node simulates at most 8ζ vertices, and the recovery in t requires at most $O(\log n)$ rounds and messages (w.h.p.), while making only $O(1)$ changes to the topology.

(b) The spectral gap of G_t is at least $\frac{(1-\lambda)^2}{8}$ where $1-\lambda$ is the spectral gap of the p-cycle expander family.

Note that staggering the inflation/deflation yields a slightly worse (but nevertheless constant) spectral gap, as stated in

Part (b) of Lemma 4. This phenomenon is caused by the additional edges (of the new p-cycle) that are added on top of the current (old) p-cycle. Since the edge expansion of our expander construction is lower bounded by the edge expansion of the old p-cycle, we can still get a constant bound on the spectral gap by invoking the Cheeger inequality (cf. Theorem 2.6 in [13]).

Implementing a Distributed Hash Table (DHT): We can leverage our expander maintenance algorithm to implement a DHT as follows: Recall that the current size s of the p-cycle is global knowledge. Thus every node uses the same hash function h_s, which uniformly maps keys to the vertex set of the p-cycle.

We consider the case where no staggered inflation/deflation is in progress and defer the more complex case to the full paper [7]: If some node u wants to store a key value pair (k, val) in the DHT, u computes the index $z := h_s(k)$. Recall that u can locally compute a shortest path $z_1, z_2, \ldots, z$ (in the p-cycle) starting at one of its simulated virtual vertices z_1 and ending at vertex z. Even though node u does not know how this entire path is mapped to the actual network, it can locally route by simply forwarding (k, val) to the neighboring node v_2 that simulates z_2; node v_1 in turn forwards the key value pair to the node that simulates z_3 and so forth. The node that simulates vertex z stores the entry (k, val). If z is transferred to some other node w at some point, then storing (k, val) becomes the responsibility of w. Similarly, for finding the value associated with a given key k', node u routes a message to the node simulating vertex $h_s(k')$, who returns the associated value to u. It is easy to see that insertion and lookup both take $O(\log n)$ time and $O(\log n)$ messages and that the load at each node is balanced.

V. Conclusion

We have presented a distributed algorithm for maintaining an expander efficiently using only $O(\log n)$ messages and rounds in the worst case and guarantee a constant spectral gap and node degrees deterministically at all times. There are some open questions: Is an $O(\log n)$ overhead sufficient for handling even a linear number of insertion/deletions per step? How can we deal with malicious nodes in this setting?

References

[1] N. Alon and J. Spencer, *The Probabilistic Method*. Wiley, 1992.

[2] G. Pandurangan, P. Raghavan, and E. Upfal, "Building low-diameter P2P networks," in *FOCS*, 2001, pp. 492–499.

[3] C. Law and K.-Y. Siu, "Distributed construction of random expander networks," in *INFOCOM 2003*, vol. 3, 2003.

[4] C. Gkantsidis, M. Mihail, and A. Saberi, "Random walks in peer-to-peer networks: Algorithms and evaluation," *Performance Evaluation*, vol. 63(3), pp. 241–263, 2006.

[5] T. Hayes, J. Saia, and A. Trehan, "The forgiving graph: a distributed data structure for low stretch under adversarial attack," *Distributed Computing*, pp. 1–18, 10.1007/s00446-012-0160-1. [Online]. Available: http://dx.doi.org/10.1007/s00446-012-0160-1

[6] A. Trehan, "Algorithms for self-healing networks," Dissertation, University of New Mexico, 2010. [Online]. Available: http://proquest.umi.com/pqdlink?did=2085415901&Fmt=2&clientId=11910&RQT=309&VName=PQD

[7] G. Pandurangan, P. Robinson, and A. Trehan, "Self-healing deterministic expanders," *CoRR*, vol. abs/1206.1522, 2012.

[8] G. Pandurangan and A. Trehan, "Xheal: localized self-healing using expanders," in *PODC '11*. ACM, 2011.

[9] J. Aspnes and U. Wieder, "The expansion and mixing time of skip graphs with applications," *Distributed Computing*, vol. 21, no. 6, pp. 385–393, 2009.

[10] R. Jacob, A. Richa, C. Scheideler, S. Schmid, and H. Täubig, "A distributed polylogarithmic time algorithm for self-stabilizing skip graphs," in *PODC '09*. ACM, 2009.

[11] D. Gillman, "A Chernoff bound for random walks on expander graphs," *SIAM J. Comput.*, vol. 27, no. 4, pp. 1203–1220, 1998.

[12] C. Scheideler, *Universal Routing Strategies for Interconnection Networks*, ser. LNCS. Springer, vol. 1390.

[13] S. Hoory, N. Linial, and A. Wigderson, "Expander graphs and their applications," *Bulletin of the AMS*, vol. 43, no. 04, pp. 439–562, 2006. [Online]. Available: http://dx.doi.org/10.1090/S0273-0979-06-01126-8

[14] V. King, J. Saia, V. Sanwalani, and E. Vee, "Towards secure and scalable computation in peer-to-peer networks," in *FOCS*, 2006.

[15] J. Augustine, G. Pandurangan, P. Robinson, and E. Upfal, "Towards robust and efficient computation in dynamic peer-to-peer networks," in *SODA*, 2012.

[16] S. Dolev and N. Tzachar, "Spanders: distributed spanning expanders," in *SAC*, 2010, pp. 1309–1314.

[17] C. Cooper, M. Dyer, and A. J. Handley, "The flip markov chain and a randomising p2p protocol," in *PODC*. ACM, 2009.

[18] M. Reiter, A. Samar, and C. Wang, "Distributed construction of a fault-tolerant network from a tree," in *SRDS 2005*, 2005.

[19] R. Melamed and I. Keidar, "Araneola: A scalable reliable multicast system for dynamic environments," *J. Parallel Distrib. Comput.*, vol. 68, no. 12, pp. 1539–1560, 2008.

[20] M. Gurevich and I. Keidar, "Correctness of gossip-based membership under message loss," *SIAM J. Comput.*, vol. 39, no. 8, pp. 3830–3859, 2010.

[21] F. Kuhn, S. Schmid, and R. Wattenhofer, "Towards worst-case churn resistant peer-to-peer systems," *Distributed Computing*, vol. 22, no. 4, pp. 249–267, 2010.

[22] M. Naor and U. Wieder, "Novel architectures for p2p applications: The continuous-discrete approach," *ACM Transactions on Algorithms*, vol. 3, no. 3, 2007.

[23] T. Hayes, N. Rustagi, J. Saia, and A. Trehan, "The forgiving tree: a self-healing distributed data structure," in *PODC '08*. ACM, 2008.

[24] J. Saia and A. Trehan, "Picking up the pieces: Self-healing in reconfigurable networks," in *IPDPS*, 2008.

[25] D. Peleg, *Distributed Computing: A Locality Sensitive Approach*. SIAM, 2000.

[26] F. Chung, *Spectral Graph Theory*. AMS, 1997.

[27] A. Lubotzky, *Discrete groups, expanding graphs and invariant measures, vol 125, Progress in Mathematics*. Birkhäuser, 1994.

[28] J. Bertrand, "Mmoire sur le nombre de valeurs que peut prendre une fonction quand on y permute les lettres qu'elle renferme." *J. l'cole Roy. Polytech. 17*, pp. 123–140, 1845.

[29] M. Mitzenmacher and E. Upfal, *Probability and Computing*. Cambridge University Press, 2005.