Efficient Bufferless Packet Switching on Trees and Leveled Networks[*]

Costas Busch[†] Malik Magdon-Ismail[‡] Marios Mavronicolas[§]

Abstract

In bufferless networks the packets cannot be buffered while they are in transit; thus, once injected, the packets have to move constantly. Bufferless networks are interesting because they model optical networks. We consider the *tree* and *leveled* network topologies, which represent a wide class of network configurations. On these networks, we study many-to-one batch problems where each node is the source of at most one packet, and the destination of an arbitrary number of packets. Each packet is to follow a preselected path from the source to the destination. Let T^* be the optimal delivery time for the packets. We have the following results:

- For trees, we present two bufferless algorithms: (i) a deterministic algorithm with delivery time $O(\delta \cdot \mathcal{T}^* \cdot \log n)$, and (ii) a randomized algorithm with delivery time $O(\mathcal{T}^* \cdot \log^2 n)$; where, δ is the maximum node degree, and n is the number of nodes. Both algorithms are distributed in the sense that packet forwarding decisions are made locally at the nodes.
- For leveled networks, we present two algorithms: (i) a centralized algorithm with delivery time $O(\mathcal{T}^* \cdot \log n)$, and (ii) a distributed algorithm with delivery time $O(\mathcal{T}^* \cdot \log^2 n)$. The first algorithm is centralized in the sense that all decisions are made by a single node. The distributed algorithm simulates the centralized one; the cost of this simulation is an extra logarithmic factor.

Our bufferless algorithms are near-optimal, and they improve on previous results for trees and leveled networks by multiple logarithmic factors.

---

[*]This paper is based on two preliminary papers: (1) "Efficient Bufferless Routing on Leveled Networks," Proceedings of the 11th International Conference on Parallel and Distributed Computing (Euro-par 2005), LNCS 3648, pp 931-940, Lisboa, Portugal, Augoust-September 2005, and (2) "Near-Optimal Hot-Potato Routing on Trees," Proceedings of the 10th International Conference on Parallel and Distributed Computing (Euro-par 2004), LNCS 3149, pp 820-827, Pisa, Italy, August-September 2004.

[†]Computer Science Department, RPI, 110 8th Street, Troy, NY 12180, USA. Email: `buschc@cs.rpi.edu`

[‡]Computer Science Department, RPI, 110 8th Street, Troy, NY 12180, USA. Email: `magdon@cs.rpi.edu`

[§]Computer Science Department, University of Cyprus, P. O. Box 20537, Nicosia CY-1678, Cyprus. Email: `mavronic@ucy.ac.cy`. projects FLAGS and DELIS.

1 Introduction

1.1 Bufferless Packet Switching

In *bufferless* networks, the nodes have no buffers for packets in transit; when a packet is received by a node, it has to be sent immediately to another node. We study *packet switching* algorithms in bufferless networks in which the task is to deliver the packets to their destinations without dropping any packets. When packets collide, i.e. two or more packets wish to follow the same link at the same time, the packets that are unable to move forward are *deflected* on alternative links. For this reason, packet switching algorithms in bufferless networks are also known as *hot-potato* or *deflection* algorithms [7]. Here we simply call them *bufferless* algorithms. Bufferless networks are interesting because they are an accurate model of optical networks in which packets are hard to buffer as they are constructed from light [45]. We consider two network architectures, trees and leveled networks.

Trees (acyclic connected graphs) are important because many real-life networks are built upon them, for example, hierarchical infrastructures. This explains the interest that trees have generated in the literature (see, for example, [2, 3, 30, 37, 41, 46, 53]). Furthermore, as articulated by Leighton [30], a spanning tree can be used to send packets in an arbitrary network.

A *leveled* network with *depth* L consists of $L + 1$ *levels* of nodes, numbered 0 to L. Every node belongs to exactly one level, and the only edges are between nodes at consecutive levels (see Figure 1). Multiprocessor network architectures such as the *butterfly* network and the *mesh* network (Figure 1) can be viewed as leveled networks, so that packet problems on these architectures are translated to routing problems on leveled networks. Other multiprocessor architectures on which packet problems are translated to leveled networks are the shuffle-exchange networks, multidimensional arrays, the hypercube, fat-trees, de Bruijn networks, and the multi-butterfly (see [18, 30] for more details). The packet problems on trees that we study here generally cannot be translated to packet problems on leveled networks.

In order to analyze bufferless algorithms, we model the network as a connected, unweighted and undirected graph with n nodes. The network is *synchronous*: time is discrete, and it takes one time step to deliver a packet across a link. At each time step, a node receives packets, and then forwards the packets to adjacent nodes. A node is allowed to send at most one packet per incident link per time step. Note that at any time step at most two packets can traverse a link, one packet in each direction of the link.

We study *many-to-one batch problems* in which we are given a set of N packets where each node is the source of at most one packet, but may be the destination of many packets. Each packet has a *preselected path* from its source to its destination. The *delivery time* of a packet switching algorithm is the time elapsed between the first packet injection until the last packet is absorbed at its destination. A packet switching algorithm that specifies how the packets move along their preselected paths is also called a *packet scheduling* algorithm. In this paper we are not concerned with how to select the paths, we are interested in how to schedule the packets given the paths.

Given the preselected paths, the delivery time depends on the *congestion* C, the maximum

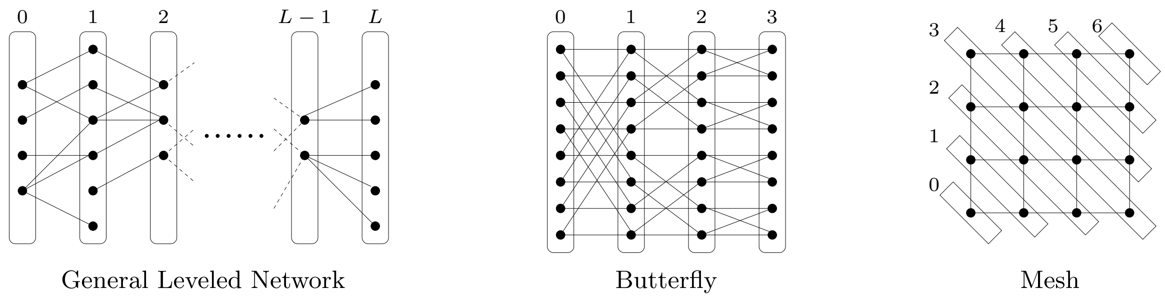

Figure 1: Leveled networks

number of packets that traverse any edge, and the *dilation* D, the maximum length of any packet path. Since at most one packet can traverse any edge at a time step, a trivial lower bound for the delivery time of any packet switching algorithm (bufferless or not) is $\Omega(C+D)$. Algorithms which give paths that minimize $C + D$ are given in [4, 5, 44, 50]. The goal of this paper is to design scheduling algorithms that will deliver the packets using the paths with delivery time close to the lower bound $C + D$. In bufferless networks, since the packets may be deflected, it may not always be possible to keep the packets on their preselected paths. We will consider algorithms in which each packet stays close to its preselected path and follows every edge in its preselected path. For such algorithms, the $\Omega(C + D)$ bound on the delivery time still holds.

For *store-and-forward* networks, in which nodes have buffers for storing packets in transit, there are packet switching algorithms for arbitrary networks whose performance is close to the $C + D$ bound [12, 31, 33, 36, 40, 43]. For arbitrary bufferless networks, a recent result by Busch *et al.* [21] shows that the packets can be delivered in time $O((C + D)\log^3(n + N))$. Here, we show that it is possible to improve this general result and obtain better delivery times for special network topologies such as trees and leveled networks, reducing the performance gap between bufferless and store-and-forward algorithms on these networks. We proceed with a more detailed discussion of our contributions.

1.2 Contribution on Trees

We present two bufferless algorithms on trees. Given a many-to-one batch problem on trees, the preselected path of each packet will be the unique shortest path in the tree from the source to the destination. Let C and D be the congestion and dilation of the preselected paths. In our algorithms, every source node determines the time at which its packet will be injected. From then on, the packet moves *greedily* to its destination, that is, whenever possible, a packet follows the link toward its destination.* In particular, we give the following algorithms:

*This is in contrast with non-greedy algorithms on which a packet may not progress to its destination even though there is an available link that takes the packet closer to its destination.

i. Tree-Deterministic with delivery time $O((\delta \cdot C + D) \log n)$, where δ is the maximum node degree in the tree. For bounded degree trees, the delivery time is optimal to within a logarithmic factor. All choices that a node makes while forwarding packets are done deterministically.

ii. Tree-Randomized with delivery time less than $\kappa(C + D) \log^2 n$, where κ is a constant. This bound holds with probability at least $1 - \frac{1}{n}$ (high probability). Randomization is used when packets select priorities; the priorities are then used to resolve packet collisions.

For bounded-degree trees, the algorithm Tree-Deterministic is better than Tree-Randomized by a logarithmic factor. However, Tree-Randomized remains near-optimal even for non-bounded degree trees.

Our algorithms are based on the idea of assigning *levels* to the nodes of the tree on the basis of *short-nodes*: a short-node r of a tree T with n nodes is a node such that if the tree were rooted at r, then each subtree contains at most $n/2$ nodes. Similarly, one can define short-nodes of r's subtrees, and so on. As we descend deeper into subtrees, the levels of the nodes increase. The level of a packet is the smallest level node that it crosses. There are $O(\log n)$ different levels.

The node levels give a natural decomposition of the tree into *inner-trees* at different levels. For every packet, the preselected path is completely within some inner-tree at a particular level. In our algorithms, the packets remain in their inner-trees until they are delivered to their destinations. Inner-trees at the same level are disjoint. So at the same level, packets of different trees can be delivered simultaneously.

Inner-trees at different levels are not disjoint. This case is handled by dividing the packets into $O(\log n)$ phases, as many phases as the number of levels. In each phase, packets of a particular level are being delivered. In the algorithm Tree-Deterministic, each phase has a duration $O(C+D)$, while in the algorithm Tree-Randomized, in order to get a high probability result, we need to allow the phases to have duration $O((C+D) \log n))$. Combining this with the bound on the number of phases, $O(\log n)$, then leads to our delivery time bounds.

The randomized algorithm uses probabilities to adjust packet priorities. Low priority packets cannot deflect higher priority packets. The packets change their priorities (from low to high) when they get deflected in a probabilistic way, so that a small number of packets is in the high priority state at the same time. Packets of high priority are unlikely to collide with each other, since their number is small. Which is why the performance of Tree-Randomized is independent of the node degree.

Our algorithms on trees are *distributed*: at every time step, each node makes packet forwarding decisions locally based only on the packets it receives at that particular time step. We assume that each source node knows the tree topology, as well as C and D for the batch problem; we emphasize, however, that a node does not need to know the specific sources and destinations of the other packets. The assumption that C and D are known is common to distributed packet switching algorithms [18, 31, 36, 40, 43].

1.3 Contribution on Leveled Networks

For a given many-to-one batch problem on a leveled network, every preselected path is monotonic in the sense that every edge in a path connects a lower level node with a node in the next higher level, i.e., a path moves from left to right on the general leveled network depicted in Figure 1.

We present two bufferless algorithms for many-to-one batch problems in leveled networks:

i. The algorithm Leveled-Centralized with delivery time $O(C \log(DN) + D)$. The delivery time is a logarithmic factor from optimal. Since $D, N \leq n$, a weaker upper bound is $O((C + D) \log n)$. The algorithm is centralized in the sense that some node has complete information about the parameters of the batch problem and schedules all packets. The algorithm computes the packet schedule in polynomial time with respect to the graph size and the batch problem parameters.

ii. The algorithm Leveled-Distributed has delivery time $O(C \log^2(DN) + D \log(DN))$ which is a logarithmic factor worse than the centralized algorithm, and a square logarithmic factor from optimal. The weaker bound $O((C + D) \log^2 n)$ holds too. This algorithm is distributed, that is, all packet forwarding decisions are made locally at the nodes. (Assuming that the nodes know the network topology and parameters C, D and N of the batch problem.)

Both results use randomization and hold with probability at least $1 - O(1/DN)$ (high probability). The distributed algorithm relies on a new technique, *reverse-simulation*, which provides an efficient distributed emulation of the centralized algorithm.

A high level description of our Algorithm Leveled-Centralized is as follows. We first divide the network into *groups* of levels, so that each group consists of $2D$ levels. The effect of this division is that each packet path belongs to exactly one group. Packets in each group are sent independently.

We now focus on one such group. We partition the group into areas of the network called *frames*. Each frame consists of roughly $\log(DN)$ levels (there are $O(D/\log(DN))$ frames in the group). The purpose of the frames is that packets are sent to the destinations by following their paths from one frame to the next. In order to achieve this, we partition the packets in the group randomly and uniformly into roughly C disjoint packet sets; each packet set creates congestion at most $\log(DN)$ with high probability. We send packets of the same set in the same frame, and packets of different sets in different frames. Thus packets in different sets do not interfere with each other and are delivered in a pipelined fashion, one set after the other in separate frames.

In a *phase*, the packets of a particular set move from one frame to the next. In order to achieve this, we use the packet dependency graph, where two packets share an edge if their paths in the frame conflict (share an edge). We color the packets in the dependency graph and send them to the next frame according to their color, so that packets of different colors do not interfere. We show that it takes $O(\log(DN))$ time to move all packets from one frame to the next (which is the phase duration).

Once a packet is injected, the time needed to deliver the packet from its source to its destination is computed by multiplying the phase duration, $O(\log(DN))$, with the number of frames which are used, $O(D/\log(DN))$, to obtain $O(D)$ time. We show that the latest possible injection time for a packet is $O(C\log(DN))$, which leads to $O(C\log(DN) + D)$ latest delivery time for any packet.

To obtain Algorithm Leveled-Distributed, we color the dependency graph in a distributed way. To accomplish this, we use a randomized distributed coloring algorithm where packets randomly pick a color, and are forwarded to the next frame according to this coloring. If the coloring is successful, then the packets can move on to the next frame; however, if the coloring is not successful (some packets collide), then the packets trace their paths backwards to the previous frame (*reverse-simulation*) and the process repeats. We show that the added inefficiency of the distributed coloring is at most one extra logarithmic factor.

1.4 Related Work

The first known form of bufferless packet switching algorithms is *hot-potato routing.* In hot-potato routing, the packets don't follow preselected paths, but rather they are sent in a greedy fashion by moving closer to the destinations. Hot-potato routing was introduced by Baran [7], and since then, hot-potato routing algorithms have been observed to work well in practice [8]. They have been used in parallel machines such as the HEP multiprocessor [48], the Connection machine [27], and the Caltech Mosaic C [47], as well as in high speed communication networks [34]. Hot-potato routing is appropriate for optical networks [1, 25, 34, 52, 54].

Hot-potato algorithms have been studied for specific network multiprocessor architectures such as the 2-dimensional mesh and torus [6, 9, 17, 19, 20, 23, 24, 28, 29, 39, 49], the d-dimensional mesh [10, 11, 15], the hypercube [14, 16, 24, 26, 42], trees [3, 22, 46], and Vertex symmetric networks [35]. (Multiprocessor architectures are extensively covered in [30].) Bhatt *et al.* [13] study hot-potato routing on leveled networks, but for different packet problems than the problems we consider here.

Most of the above papers study special cases of many-to-one batch problems, such as permutation problems or random-destination problems. In these batch problems the preselected paths are not given, only the sources and destinations of the packets are specified. The paths are found dynamically while the packets are delivered in the network. Usually, such algorithms cannot give good paths for the packets resulting in delivery times that are only worst-case optimal (not optimal for every instance of the batch problem they solve).

Our algorithms can be used to solve such kinds of batch problems too. For such a batch problem, we first get good paths with $C + D$ close to optimal (using known results that give good paths in [4, 5, 44, 50]). Then we apply our algorithms using the good paths, to obtain delivery time within logarithmic factors from optimal, for *any* instance of the batch problem (and not only for worst cases). Also our algorithms are applicable to *arbitrary* many-to-one batch problems, and not only to special cases of batch problems.

1.4.1 Related Work on Trees

Various bufferless packet switching models for trees have been considered. *Matching routing* on trees is considered in [2, 41, 53]; here, at each time step, a set of edges with disjoint endpoints is chosen, and then the packets at the endpoints of each selected edge are exchanged. All of the results in matching routing consider permutation batch problems (without preselected paths) and provide algorithms with delivery time $O(n)$, where n is the number of packets. The results are only worst-case optimal and not optimal for any case, while ours are every case near-optimal (given good paths).

In [3, 22, 51], the *direct routing* model is considered on trees; here, an injection time schedule is computed such that the packets follow shortest paths to their destinations without collisions. Direct routing algorithms are *centralized*, i.e., some central node has global information about the batch problem and computes the injection times of all the packets. In contrast, the bufferless algorithms for trees that we consider here are distributed and rely on deflections. In [3, 51] direct routing algorithms with delivery time $O(n)$ are given, for permutation batch problems with n packets. Again, this result is worst-case optimal, while ours is every case near-optimal. In [22], a direct routing algorithm (on trees) with optimal $O(C + D)$ delivery time is presented; however, as already mentioned, direct routing algorithms are centralized while our algorithms are distributed.

Roberts *et al.* [46] consider greedy hot-potato routing and show that there exist permutation batch problems for which any greedy hot-potato algorithm requires $\Omega(n)$ delivery time. However, there exist simple permutation batch problems with asymptotically smaller delivery time (for example in a star network topology).

1.4.2 Related Work on Leveled Networks

For leveled networks, the most related work to ours is [18], which gives a distributed algorithm for leveled networks with bound $O((C + L)\log^9(LN))$. We improve this result by seven logarithmic factors; moreover, that result was expressed in terms of L instead of D which we consider here. The algorithm in [18] shares similarities with our algorithm in the sense that the network is partitioned into frames and the packets move along the frames to the destinations.

A recent result in [21] shows that it is possible to obtain a general bufferless packet switching algorithm for arbitrary networks with delivery time $O((C + D)\log^3(n + N))$. However, that result doesn't take advantage of the special structure of leveled networks, which allows us to obtain a smaller delivery time, by one or two logarithmic factors.

Leveled networks have also been studied in the context of store-and-forward scheduling (with buffers), by Leighton *et al.* [32], where they present an $O(C + L + \log N)$ randomized algorithm with constant size buffers. For store-and-forward scheduling, there have been many results on obtaining optimal $O(C+D)$ algorithms for arbitrary networks [31, 33, 36, 40, 43]. Our results reduce the gap between the performance of bufferless algorithms and store-and-forward algorithms for trees and leveled networks.

Outline of the Paper

In the paper, we proceed as follows. In Section 2 we present our results on trees. In Section 3 we present our results on leveled networks. We conclude with a discussion in Section 4.

2 Trees

In this section we present our results on trees. We start with some necessary preliminaries in Section 2.1, where we describe the decomposition of a tree into inner-trees. In the same section we describe other details, such as how the deflections are handled in our algorithms. We then continue with the description and analysis of the deterministic algorithm in Section 2.2, and the randomized algorithm in Section 2.3.

2.1 Preliminaries on Trees

A tree $T = (V, E)$ is a connected acyclic graph with $|V| = n$ and $|E| = n - 1$. The *degree* of node v is the number of nodes adjacent to v. Let $v \in V$; then, T induces a subgraph on $V - \{v\}$ which consists of a number (possibly zero) of connected components. Each such connected component is a *subtree of* v *in* T.[†] If v is adjacent to k nodes in T, then there are k disjoint subtrees $T_1, \ldots, T_k$ of v, one for each node $v_i \in T_i$ that is adjacent to v. The *distance* from v to u, is the number of edges in the (unique) shortest path from v to u. We continue to examine various properties of trees and how packets are sent on them.

2.1.1 Inner-Trees

The main idea behind our algorithms is to look at the tree from the point of view of a short node (see Figure 2). A node v in the tree is *short* if every subtree of v contains at most $n/2$ nodes. At least one short node is guaranteed to exist. (Algorithm Find-Short-Node in the Appendix finds one in $O(n)$ time.)

We now define recursively the *level* ℓ of a node, and the *inner-trees* of T as follows. The tree T is the only inner-tree at level $\ell = 0$. The only node at level $\ell = 0$ is the short node of T (we pick one of the short nodes of T). Assume we have defined inner-trees up to level $\ell \geq 0$. Every connected component obtained from the inner-trees of level ℓ by removing the short nodes of these inner-trees at level ℓ is an inner-tree at level $\ell + 1$. The level $\ell + 1$ nodes are precisely the short nodes of the inner-trees at level $\ell + 1$. The process is illustrated in Figure 3. (We can easily construct an $O(n^2)$ procedure to determine the node levels and inner-trees using the Algorithm Find-Short-Node in the Appendix.)

From the above definition we immediately obtain the following properties: (*i*) every inner-tree is a tree, (*ii*) the maximum level of any node and inner-tree is no more than $\log n$, (*iii*) an inner-tree T' at level ℓ contains a unique node x at level ℓ, which is the short node

[†]Note that for unrooted trees which we consider here, a subtree of a node v originates from every adjacent node of v; in contrast, the convention for rooted trees is that a subtree of v is any tree rooted at a child of v.

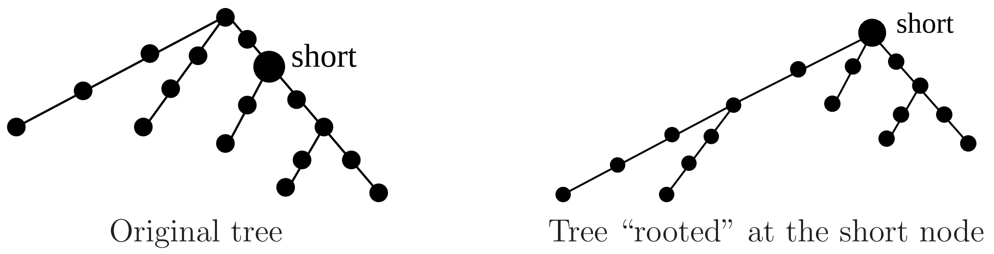

Figure 2: The short node

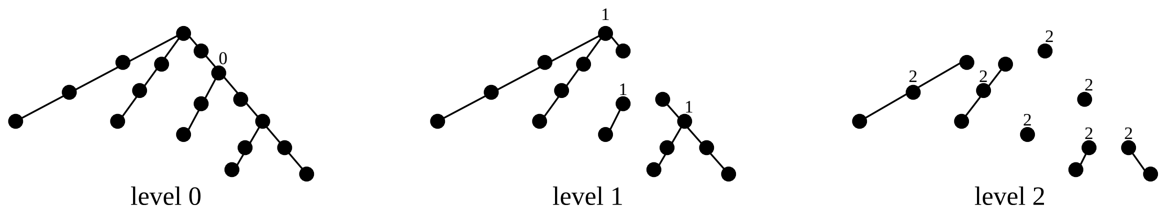

Figure 3: The process of constructing inner-trees at levels 0, 1 and 2

of the inner-tree (we say that x is the *inducing* node of T'), (iv) any two inner-trees at the same level are disconnected, and (v) all nodes in a level-ℓ inner-tree other than the inducing node have a level that is smaller than ℓ.

2.1.2 Paths in Trees

A *path* is any sequence of nodes $(v_1, v_2, \dots, v_k)$, where $(v_i, v_{i+1}) \in E$, for all $1 \le i \le k-1$. The length of the path is the number of its edges. The preselected path of a packet π is the shortest path from the source to the destination node of the packet. We also refer to this path as the *original* path.

In our algorithm we will consider shortest paths on trees. Let ℓ be the minimum level of any node in the original path of π. Then, there is a unique node v with level ℓ in the path of π (since otherwise inner-trees of the same level would not be disconnected). Let T' be the inner-tree that v is inducing. The whole original path of π must be within T' (from the definition of inner-trees). We say that the level of packet π is ℓ, and that the inner-tree of π is T'. Note that all the packets of level ℓ and inner-tree T' cross the same (unique) node v at level ℓ in T'. For example, see Figure 4 and inner-tree T'_1, which is rooted at the node of level ℓ, and all the paths cross the root.

Since inner-trees at the same level are disjoint, the packets paths on different inner-trees of the same level do not intersect, that is, they have no common node and edge. This observation is very useful for our algorithms, because we will deliver and maintain the packets inside the inner-trees they belong to. Thus, packets on different inner-trees at the same level will be delivered simultaneously. For example, two different trees T'_1 and T'_2 of level ℓ are depicted in Figure 4. Since the trees are disjoint, paths on different trees do not intersect.

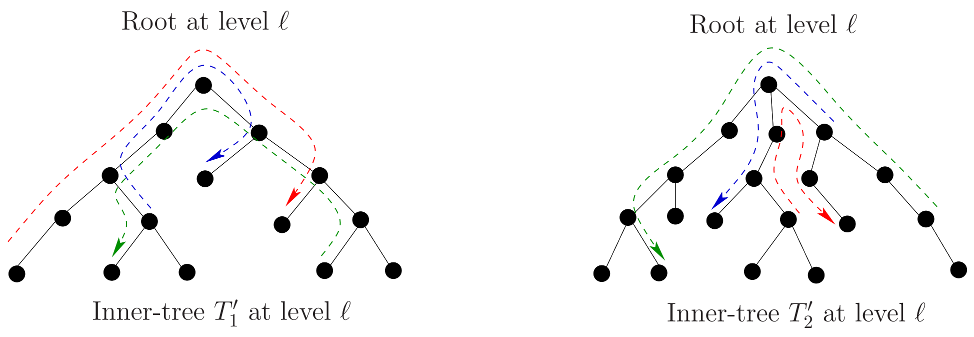
Root at level ℓ
Inner-tree T'_1 at level ℓ
Root at level ℓ
Inner-tree T'_2 at level ℓ

Figure 4: Packet paths on two disjoint inner-trees T'_1 and T'_2 at the same level ℓ

Assume now that packet π is injected into the network. At any time step t, the *current path* of a packet is the shortest path from the current node that the packet resides at until the destination node. At the moment that the packet is injected, its current path is its original path. At any time step, a packet either moves *forward* (closer to the destination) or it is deflected. When the packet moves forward, its current path gets shorter by removing the edge that the packet follows. Any time that the packet is deflected, its current path grows by the edge on which it was deflected.

In particular, consider a packet π with preselected path p which has path length $|p|$. Let $p(t)$ denote the current path at time t. At time 0, the current path is the preselected path, $p(0) = p$. Suppose that at time t, packet π is in node v_i, with current path $p(t) = (v_i, v_{i+1}, \ldots, v_k)$. If at time t, packet π successfully follows the first edge (v_i, v_{i+1}) in $p(t)$ (the packet moves forward), then, at time $t+1$, packet π appears in node v_{i+1} with current path $p(t+1) = (v_{i+1}, \ldots, v_k)$. On the other hand, if at time $t+1$ packet π is deflected toward a node v_j, then at time $t+1$ it appears in node v_j with current path $p(t+1) = (v_j, v_i, v_{i+1}, \ldots, v_k)$. Thus, if the packet moves forward, $|p(t+1)| = |p(t)| - 1$ and if it is deflected, then $|p(t+1)| = |p(t)| + 1$.

The deflections may take a packet on edges that was not in the original path. Thus after time 0, the current path may be different than the original path. However, a packet will traverse all the the edges of its preselected path before it is delivered to the destination. Note that in trees the current path is always the shortest path from the current node to the destination node.

2.1.3 Canonical Injections and Deflections

In our algorithms, a packet remains in its source node until a particular time step at which the packet becomes *active*. When the packet becomes active, it is injected at the first available time step on which the first link of its original path is not used by any other packets that reside at its source node. We call such an injection a *canonical injection*.

After the packets are injected they move toward the destination by following their current paths. Two or more packets may *meet* if they appear in the same node at the same time step. We say that two or more packets *collide* if they meet at some time step wish to follow the same link forward. In a collision, one of the packets will successfully follow the link, while the other packets must be deflected. In a *greedy* algorithm, a packet always attempts to follow its forward link unless it is deflected by another packet with which it collides for the same edge. The algorithms we consider for trees here are greedy.

In our algorithms, packets are deflected in a particular fashion so as to ensure that the congestion of the edges on the current paths never increases more that the congestion of the preselected paths. Consider a node v at time step t. Let S_f denote the set of packets which moved forward in the previous time step $t-1$, and now appear in v at time step t. Let E_f be the set of edges that the packets in S_f followed at time $t-1$. Let π be a packet in node v that will be deflected at time t. Node v first attempts to deflect π along an edge in E_f. If this fails (due to other packets that move forward using edges in E_f) any other edge adjacent to v is used for the deflection. Thus, π is deflected on E_f unless other packets use all the edges in E_f. We call this process of deflecting packets *canonical deflection*.

If π successfully follows an edge in E_f, then we say that the deflection is *safe*. We will show that in our algorithms when the deflections are canonical then they are also safe. Safe deflections have the following effect. Let e be the edge of E_f that π will be deflected on. Let σ be the packet of S_f that followed e at time step $t-1$ toward v. Then, the edge e is transferred from the current path of σ to the current path of π; thus, the edges "recycle" from one current path to another. We now show that when injections and deflections are canonical, the deflections are always safe.

Lemma 2.1 *If packet injections and deflections are canonical, then packet deflections are also safe.*

Proof: Let v be some node, and S the set of packets that will be sent from v at time step t. We write $S = S_f \cup S_d \cup S_i$, where S_f, S_d and S_i are disjoint sets such that: S_f are those packets which moved forward at time step $t-1$, in order to appear in v at time step t; S_d are those packets that were deflected at time step $t-1$; S_i are those packets which are injected at time step t in node v. Let E_f and E_d denote the sets of edges adjacent to v which the packets in S_f and S_d followed respectively, at time step $t-1$. Clearly, $|S_f| = |E_f|$ and $|S_d| = |E_d|$; furthermore, since $S_f \cap S_d = \emptyset$, it must be $E_f \cap E_d = \emptyset$. Let S' denote the set of packets of S that will be deflected. We only need to show that the packets of S' follow edges of E_f.

We can write $S_f = S_1 \cup S_2 \cup S_3 \cup S_4$, where S_1 are packets that will move forward on edges of E_f, S_2 are packets that will move forward on edges of E_d, S_3 are packets that will move forward on edges not in $E_f \cup E_d$, and S_4 are packets that will be deflected; sets S_1, S_2, S_3, S_4 are disjoint. Furthermore, we can write $S_d = S_5 \cup S_6$, where S_5 are packets of S_d that will move forward on edges of E_d and S_6 are packets that will be deflected; sets S_5 and S_6 are disjoint. Clearly, $S' = S_4 \cup S_6$.

For every packet of S_f which moves forward on an edge of E_d, a packet of S_d must be deflected. This implies that $|S_2| = |S_6|$. Let A be the set of edges of E_f that are not used by packets of S_1; in other words, A is the set of edges of E_f on which safe deflections can

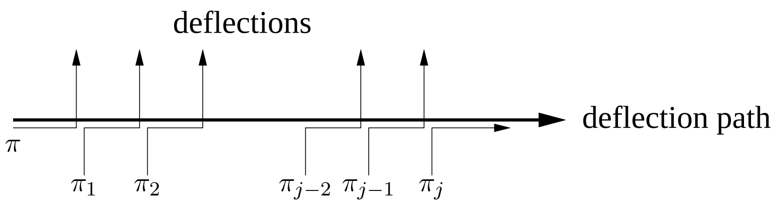

Figure 5: The deflection sequence $\pi_1, \pi_2, \ldots, \pi_j$.

occur. We have that $|A| = |S_f| - |S_1|$. We also have that $|S'| = |S_4| + |S_6| = |S_4| + |S_2|$. Equivalently, $|S'| = |S_f| - |S_1| - |S_3|$. It follows that $|S'| \leq |A|$. Subsequently, all packets can be deflected on edges of E_f. It follows that all deflections if made canonically are also safe, concluding the proof. ∎

Consider some edge e. The congestion of edge e at time t, denoted C_e^t, is the number of current paths that go through edge e at the beginning of time step t. Let $C^t = \max_{e \in E} C_e^t$, namely, C^t denotes the network congestion at time t. Note that $C = C^0$. Safe deflections imply that for any edge e and any time step t, C_e^t is no more than C_e^0, since edges are transferred from one current path to another one due to deflections, and the number of original paths crossing e is C_e^0. Therefore, from Lemma 2.1 we obtain:

Lemma 2.2 *If packet injections and deflections are canonical, then $C^t \leq C$, for any $t \geq 0$.*

2.1.4 Deflection Sequences

In the analysis of Algorithm Tree-Deterministic, we will use a technique developed by Borodin *et al.* [15, Section 2], called a "general charging scheme", with which they analyze deflection algorithms. Below, we adapt the discussion from [15, Section 2] so that it is appropriate for trees. Consider a packet π that was deflected at time t_1 by packet π_1. Define a *deflection sequence* and a *deflection path* with respect to this deflection as follows. Follow packet π_1 starting at time t_1 either to its destination or up to time $t_2 > t_1$, when it is deflected for the first time after t_1 by some packet π_2. Follow π_2 from time t_2 either to its destination or until some other time $t_3 > t_2$, when π_2 is deflected for the first time after t_2 by some packet π_3. Continue in the same manner until a packet π_j is followed to its destination. Define the sequence of packets: $\pi_1, \pi_2, \ldots, \pi_j$ as the deflection sequence of π at time t_1. Define the path that follows this sequence of packets from the point of deflection to the destination of π_j to be the deflection path. (See Figure 5.)

Claim 2.3 [15] *Suppose that for any deflection of packet π from node v to node u, the shortest path from node u to the destination of π_j (the last path in the deflection sequence) is at least as long as the deflection path. Then, π_j cannot be the last packet in any other deflection sequence of packet π.*

Claim 2.3 can be applied for greedy packet scheduling on trees. Claim 2.3 implies that we can "charge" the deflection of π to packet π_j, in the sense that when a packet is deflected another packet makes it to the destination. This implies the following corollary.

Corollary 2.4 [15] *If the deflection sequence for each of the deflections incurred by a packet switching algorithm satisfies the conditions of Claim 2.3, then the arrival time of each packet is bounded by* $dist(\pi) + 2(k-1)$*, where* $dist(\pi)$ *is the length of the shortest path from the source of packet* π *to its destination and* k *is the number of packets.*

2.2 A Deterministic Algorithm on Trees

We present the Algorithm Tree-Deterministic (Algorithm 1). Each node is the source of at most one packet. The algorithm is described in terms of the actions of a packet π. Let v be the source of π. Node v first computes the level ℓ of the packet. Then according to the packet's level, node v makes π active at a particular time step that corresponds to the delivery of packets at level ℓ. Then packet π is injected canonically and moves greedily in the network until it is absorbed at its destination. All deflections are canonical.

In the analysis of our algorithm, we will show that once the packet is injected it will remain in its inner-tree until it is delivered to its destination. Thus, a packet will interfere (i.e. collide) only with packets of the same inner-tree. Consequently, packets at different inner-trees at the same level can be delivered simultaneously. Only packets at different levels can interfere with each other. In order to avoid this kind of interferences, packets at different levels are sent separately into phases, where phase i corresponds to the delivery of packets at level i. Thus, in Algorithm Tree-Deterministic a packet π becomes active at the beginning of the ℓth phase, and it is delivered before the end of the phase.

```
Algorithm: Tree-Deterministic
Input: A tree $T$ of maximum node degree $\delta$; A set of packets $\Pi$ with preselected
       paths having congestion $C$ and dilation $D$; Each node is the source of at most
       one packet; Each node knows $T, C, D$;
Do for each packet $\pi$ of level $\ell$:
begin
1   Packet $\pi$ gets active at time $\tau \cdot \ell$ (start of $\ell$th phase), where $\tau = 2(\delta \cdot C - 1) + D$
    (duration of phase);
2   The injection and deflections of packet $\pi$ are canonical;
3   Packet $\pi$ moves greedily to its destination;
end
```

Algorithm 1: Tree-Deterministic

2.2.1 Analysis of Deterministic Algorithm

We proceed with the detailed analysis of the algorithm. Lemma 2.1 implies that all deflections are safe. Let m be the maximum level in T (note that $m \leq \log n$). We divide time into consecutive phases $\phi_0, \phi_1, \ldots, \phi_m$, such that each phase consists of τ time steps. Write $\Pi = \Pi_0, \Pi_1, \ldots, \Pi_m$, where Π_i are packets of level i. From the algorithm, the packets of set Π_i become active at the first time step of phase ϕ_i. We will show that all packets of level i are absorbed during phase ϕ_i. In particular, we will show that the following invariants hold, where $i \geq 0$:

P_i: all packets of $\Pi_0 \cup \Pi_1 \cup \cdots \cup \Pi_i$ are absorbed by the end of phase ϕ_i.

In order to show that the properties P_i are indeed invariants, we will first show that the following induction hypothesis holds, where $i \geq 0$, and P_{-1} is taken to be true by default:

Q_i: if P_{i-1} holds, then all packets of Π_i are absorbed by the end of phase ϕ_i.

Now, we will consider a particular level $\ell \geq 0$ and phase ϕ_ℓ. Assume that $P_{\ell-1}$ holds (namely, all packets of $\Pi_0 \cup \Pi_1 \cup \cdots \cup \Pi_{\ell-1}$ have been absorbed by the end of phase $\phi_{\ell-1}$). We will show that Q_ℓ holds; in other words, we will show that all packets of Π_ℓ will be absorbed by the end of phase ϕ_ℓ. Notice that in phase ϕ_ℓ the only packets injected are those of Π_ℓ. So, from now on, we will consider phase ϕ_ℓ and only the packets Π_ℓ. We will show that each packet remains inside its inner-tree for the entire duration of ϕ_ℓ. (Note that an inner-tree can be connected with another inner-tree of lower level.)

Lemma 2.5 *During phase ϕ_ℓ, each packet of Π_ℓ remains inside its inner-tree.*

Proof: Assume for contradiction that some packet of Π_ℓ leaves its inner-tree during phase ϕ_ℓ. Let π be the first packet which leaves its inner-tree, and let t be the time step at which this event occurs. That is, at time step t, packet π appears in a node v which is not in its inner-tree, and at time step $t-1$, packet π was in a node u in its inner-tree. Thus, in node u and time $t-1$, packet π is deflected, since the destination of π is in its inner-tree. Since deflections are safe, there must be another packet σ that moved forward from node v to node u at time step $t-2$. Since inner-trees of the same level are disjoint, we have that packet σ left its inner-tree before packet π, a contradiction. ∎

From Lemma 2.5, it follows that only packets of the same inner-tree meet with each other; thus, only packets of the same inner-tree may collide with each other. From now on, we will consider only packets of some particular inner-tree T' of level ℓ, and denote the level-ℓ inducing node of T' by r. Next, we show that every packet with inner-tree T' will be absorbed in phase ϕ_ℓ.

Corollary 2.4, applies to Algorithm Tree-Deterministic. For any packet π, we have that $\text{dist}(p) \leq D$. Moreover, at the beginning of phase ϕ_ℓ, the number of packets in inner-tree T' does not exceed $\delta \cdot C$, since: (i) the original path of each packet of T' goes through node r, (ii) the degree of r is at most δ, and (iii) each edge adjacent to r has congestion $C^{\tau \cdot \ell} \leq C$

(a consequence of Lemma 2.2). Further, no more packets can be added in T' during phase ϕ_ℓ. Thus, from Corollary 2.4, all packets in inner-tree T' will be absorbed within a period of time $2(\delta \cdot C - 1) + D = \tau$. Subsequently, all packets of inner-tree T' are absorbed by the end of phase ϕ_ℓ. This implies that all packets of Π_ℓ are absorbed by the end of phase ϕ_ℓ. Therefore, we have shown the following lemma:

Lemma 2.6 *Q_ℓ holds for all $\ell \geq 0$.*

Since P_{-1} holds, by induction, we have the following result.

Lemma 2.7 *P_ℓ holds for all $\ell \geq 0$.*

Lemma 2.7 implies that P_m holds. The fact that P_m holds further implies that all packets will be absorbed by the end of phase ϕ_m. Since $m \leq \log n$, all packets are absorbed by time step $\tau \cdot (m+1)$ which is at most $(2(\delta \cdot C - 1) + D)(\log n + 1)$. We have the following theorem and its immediate corollary:

Theorem 2.8 *The delivery time of Algorithm* Tree-Deterministic *is bounded by $O((\delta \cdot C + D) \log n)$.*

Corollary 2.9 *If δ is bounded by a constant, then the delivery time of Algorithm* Tree-Deterministic *is bounded by $O((C + D) \log n)$.*

2.3 A Randomized Algorithm on Trees

Here, we present the Algorithm Tree-Randomized (Algorithm 2). The difference between Algorithms Tree-Randomized and Tree-Deterministic is that the packets have now priorities. As we will show in the analysis, the use of priorities removes the dependence on the node degrees but increases the delivery time by a logarithm, due to randomization.

There are two levels of priority: low and high. At any time step, a packet is in one of these two priorities. A packet of high priority has precedence over a packet of low priority in a collision. Collisions between packets of the same priority are resolved arbitrarily in a canonical fashion. Initially, when a packet becomes active, it has low priority. The packet may change its priority only after a collision. If a packet is deflected, its priority is set to high with probability p (where p is specified in the algorithm), and to low with probability $1 - p$, independent of its previous priority. In the analysis, we will show that a packet with high priority will reach its destination without being deflected (with high probability).

2.3.1 Analysis of Randomized Algorithm

We now proceed with the analysis of the algorithm. Lemma 2.1 implies that all deflections are safe. Let m be the maximum level in T (note that $m \leq \log n$). We divide time into consecutive phases $\phi_0, \ldots, \phi_m$, and the packets into different sets $\Pi_0, \ldots, \Pi_m$, as we did in Section 2.2. We also consider the properties P_i for $0 \leq i \leq m$, as defined in Section 2.2. We

```
Algorithm: Tree-Randomized
Input: A tree $T$; A set of packets $\Pi$ with preselected paths having congestion $C$ and
       dilation $D$; Each node is the source of one packet; Each node knows $T, C, D$;

Do for each packet $\pi$ at level $\ell$:
begin
1    Packet $\pi$ gets active at time step $\tau \cdot \ell$ (start of $\ell$th phase), where $\tau = 16 \cdot (C + D) \cdot (2 \log n + \log\log 2n) + 3D + 1$ (duration of phase);
2    The injection and deflections of packet $\pi$ are canonical;
3    Packet $\pi$ moves greedily to its destination;
4    When packet $\pi$ becomes active it has low priority;
5    If $\pi$ is deflected at time step $t$, then on the next time step $t+1$, the priority of $\pi$
     becomes high with probability $p = 1/(4(C+D))$, and low with probability $1-p$
     (no matter what the previous priority was). The packet preserves the new priority
     until the next deflection;
end
```

Algorithm 2: Tree-Randomized

will show that properties P_i hold with high probability. In order to do this, we will first show that if P_{i-1} holds for any particular $i \geq 0$ then P_i holds with high probability (a probabilistic version of Q_i as defined in Section 2.2.

Now, we consider a particular level $\ell \geq 0$ and phase ϕ_ℓ. Let $t_1, t_2, \ldots, t_\tau$ denote the time steps of phase ϕ_ℓ. Assume that $P_{\ell-1}$ holds (namely, all packets of $\Pi_0 \cup \Pi_1 \cup \cdots \cup \Pi_{\ell-1}$ have been absorbed by the end of phase $\phi_{\ell-1}$). We will show that Q_ℓ holds with high probability; namely, we will show that all packets of Π_ℓ will be absorbed by the end of phase ϕ_ℓ with high probability. Notice that in phase ϕ_ℓ the only packets injected are those of ϕ_ℓ. So, we will consider only the packets Π_ℓ. Notice that Lemma 2.5 holds. Thus, from now on, we will consider only packets of some particular inner-tree T' of level ℓ, and denote the level-ℓ inducing node of T' by r. We will show that every packet with inner-tree T' will be absorbed in phase ϕ_ℓ, with high probability. Let $T_1, T_2, \ldots, T_w$ denote the subtrees of r in T'. We first show some interesting properties about these subtrees.

Lemma 2.10 *The number of level-ℓ packets with destinations in T_j, for $1 \leq j \leq w$, is at most C.*

Proof: Let e denote the edge that connects T_j with node r. All the level-ℓ packets with destination in T_j use e, and since the edge congestion never increases (Lemma 2.1), there can be at most C such packets. ∎

Lemma 2.11 *Consider any time step t_i, $1 \leq i \leq \tau$, and any subtree T_j, $1 \leq j \leq w$. The number of packets that appear in T_j at time step t_i is at most C.*

Proof: Let A denote the set of packets with sources in T_j and B the set of packets with destinations in T_j. Let e be the edge that connects tree T_j with r. It must be that $|A|+|B| \leq C$, since all the packets in A and B have edge e on their original path, and the congestion can not exceed C.

Let X_i denote the set of packets which appear in T_j at time step t_i. We can write $X_i = Y_i \cup Z_i$, where Y_i are packets with destinations outside T_j, and Z_i are packets with destinations in T_j. We know that $Y_1 = A$. For $i > 1$, we can write $|Y_i| = |A| + a - b$, where a is the number of packets which entered T_j, and b is the number of packets which left T_j, between time steps t_1 and t_i, and all these packets have destinations outside T_j. Consider a packet π with destination outside T_j, which enters T_j in time step t_i (i.e. packet π traverses e at time step t_{i-1}). It must be that packet π has entered the network due to a deflection. Since deflections are safe, it must be that another packet $\sigma \in Y_{i-2}$ followed edge e forward at time step t_{i-2} (i.e. packet σ has its destination outside T_j). Thus, for any packet similar to π that enters T_j, there is another similar to σ the leaves T_j. This implies that $a \leq b$. Therefore, $|Y_i| \leq |A|$. Moreover, we know that $Z_i \subseteq B$. Which implies that $|X_i| = |Y_i| + |Z_i| \leq |A| + |B| \leq C$, as needed. ■

We define the *depth* of a node v, as the distance of the node from r, and the depth of a packet as the depth of the node in which it appears.

Lemma 2.12 *At time step t_i, $1 \leq i \leq \tau$, packets in subtree T_j, $1 \leq j \leq w$ have depth $\leq D$.*

Proof: We will show a stronger result: at time step t_i, packets in subtree T_j have depth $\leq D$, *and* packets at level D are in *isolation*, i.e., no more than one depth-D packet appears in the same node. We prove the claim by induction on i.

For $i = 1$, the claim holds trivially, since every node is the source of one packet which is injected in isolation at time step t_1; moreover, the original path dilation does not exceed D. Assume that the claim is true for any time step t_i, where $1 \leq i < k \leq \tau$ and consider time step t_k. Note that the destination node of any packet has depth at most D (since the length of the original paths are at most D and all these paths cross node r). From the induction hypothesis, at time step t_{k-1}, all packets have depth D or lower. Consider the packets at depth D at time step t_{k-1} (by the induction hypothesis, these packets are in isolation). It must be that these packets wish to move to depth $D-1$, since none of them have reached their destinations, and all of them have destinations at depth D or lower. All these packets successfully follow the links toward depth $D-1$, at time step t_k. Therefore, at time step t_k, no packet will have depth greater than D. Moreover, at time step t_k the packets at depth D can only be packets which had depth $D-1$ at time step t_{k-1} (since, from the induction hypothesis, there are no packets at depth $D+1$ at step t_{k-1}). These packets will appear in isolation at depth D on time step t_k, since each of these packets follows a different edge leading to depth D. Thus the claim holds for time step t_k, and the lemma follows by induction. ■

Let $R = [t_a, t_b]$, where $1 \leq a \leq b \leq \tau$, denote a time period containing time steps $t_a, t_{a+1}, \ldots, t_b$.

Lemma 2.13 *Consider a time period $R = [t_a, t_b]$, $1 \le a \le b \le \tau$, and a tree T_j, $1 \le j \le w$. The number of different packets that appeared in T_j during period R are at most $C + b - a$.*

Proof: From Lemma 2.11, we know that the number of packets that appear in T_j at time step t_a are at most C. At any subsequent time step, at most one new packet enters subtree T_j, which implies that during period R, the number of different packets that appeared in T_j is at most $C + b - a$. ■

We can bound the number of different packets that a packet π may collide with in a period as follows:

Lemma 2.14 *Consider a time period $R = [t_a, t_b]$, $1 \le a \le b \le \tau$, in which a packet π is not deflected. During period R packet π may have collided with at most $2C + b - a$ different packets.*

Proof: Assume that at time step t_a, packet π is in subtree T_j and wishes to move to subtree T_k, where its destination resides, so that $k \ne j$. (If π has destination node r, or at time step t_a is either in r or T_k, then the analysis is similar.) Assume that packet π resides in subtree T_j for period $R' = [t_a, t_c]$, where $1 \le a \le c < b$. In order for π to collide with some packet σ in T_j, it must be that packet σ resides in T_j during period R'. From Lemma 2.13, the number of packets similar to σ is at most $C + c - a \le C + b - a$.

In time period $[t_{c+1}, t_b]$, packet π follows a path that includes the node r and a path in the subtree T_k. At the nodes of this path, packet π may collide only with packets that have destinations in T_j. From Lemma 2.10, the number of these packets is at most C. Therefore, the total number of different packets that π may collide with during period R is at most $2C + b - a$. ■

Consider a time period $R = [t_a, t_b]$ in which packet π is not deflected. From Lemma 2.14, it follows that during period R, packet π may collide with at most $2C + b - a$ packets. Let σ be any such packet. It is easy to see that σ will collide at most once with π during period R (otherwise, packet π and σ would meet at two nodes at two different time steps during R, and this would imply that there are two different paths connecting the two nodes, which is impossible). Using this observation, we now prove:

Lemma 2.15 *Consider a time step t_i, where $1 \le i \le \tau - 2D$, at which packet π is in high priority. The probability that packet π reaches its destination in subsequent time steps without deflections is at least $1/2$.*

Proof: From Lemma 2.12, π has depth at most D. The destination of π also has depth at most D (since the original paths have length at most D and cross node r). Hence, at time step t_i the current path π has length at most $2D$. Now, consider time period $R = [t_i, t_{i+2D-1}]$. If during R packet π is not deflected (including time step t_{i+2D-1}), then it successfully reaches its destination node.

Since packet π has high priority, it can be deflected only by other packets of high priority. Any other packet σ has only one chance to deflect packet π. This chance is given to packet σ with probability at most p: first packet σ increases its priority with probability p on its last deflection, and then it is on a collision course with packet π. From Lemma 2.14, we have that the number of packets in a similar situation to that of σ is at most $2C+i+2D-1-i = 2C+2D-1 \leq 2(C+D)$. Therefore, the probability that packet π will be deflected by any of these packets is at most $2(C+D)p = 2(C+D)/(4(C+D)) = 1/2$. Thus, with probability at least $1/2$, no packet will deflect packet π. ■

Using Lemma 2.15, we obtain:

Lemma 2.16 *It a packet π gets deflected at time step t_i, $1 \leq i \leq \tau - 2D - 1$, then the probability that in subsequent time steps packet π reaches the destination node without deflections is at least $p/2$.*

Proof: After the packet is deflected at time step t_i, it becomes a high priority packet at time step t_{i+1} with probability p. From Lemma 2.15, we know that packet π is not deflected until it reaches its destination with probability at least $1/2$. Thus, after the deflection, packet π will have high priority and will reach its destination without deflections with probability at least $p/2$. ■

From Lemma 2.16, we have that every time a packet is deflected, it has a chance to increase its priority and reach its destination without deflections. We next estimate how many times a packet gets deflected in a particular time period.

Lemma 2.17 *Consider a packet π which is in the network for the entire time period $R = [t_1, t_x]$, where $D \leq x \leq \tau$. Packet π gets deflected at least $(x-D)/2$ times in period R.*

Proof: Let a denote the number of times that π moves forward and b the number of times it is deflected, up to (and including) time step t_{x-1}, then $a + b = x - 1$. Every time that the packet moves forward its distance to the destination decreases, while every time it moves backward the distance increases. Let d_x denote the distance of π from its destination at time step t_x. We have that $d_x = d_1 - a + b$. Equivalently, $d_x = d_1 - x + 2b + 1$, which implies: $b = (d_x - d_0 + x - 1)/2$. We know that $d_x \geq 1$ (since π is in the network at time step t_x), and that $d_1 \leq D$, since in the original path of π the distance from its destination is at most D. Thus, $b \geq (x-D)/2$. ■

Next we compute the probability that packet π reaches its destination in phase ϕ_ℓ.

Lemma 2.18 *Packet π reaches its destination in phase ϕ_ℓ with probability at least $1 - 1/(n^2 \log 2n)$.*

Proof: Consider the time period $R = [t_1, t_{\tau-2D-1}]$, and suppose that π did not reach its destination yet. From Lemma 2.17, we have that π is deflected at least $x = (\tau - 2D - 1 - D)/2 = 8(C + D)(2 \log n + \log \log 2n)$ times in period R. From Lemma 2.16, it follows that every time the packet is deflected in period R it has probability at least $p/2$ to reach its destination without further deflection. In other words, packet π fails to reach its destination without deflection with probability at most $1-p/2$. Therefore, π fails to reach its destination after x deflections with probability at most $(1 - p/2)^x$.[‡] We have that,

$$\left(1 - \frac{p}{2}\right)^x = \left(1 - \frac{1}{8(C + D)}\right)^{8(C+D)(2\log n+\log\log 2n)} \le \frac{1}{e^{2\log n+\log\log 2n}} = \frac{1}{n^2 \log 2n}.$$

Thus, packet π reaches its destination in phase ϕ_ℓ with probability at least $1 - 1/(n^2 \log 2n)$.
■

Now, we consider all packets Π_ℓ in phase ϕ_ℓ.

Lemma 2.19 *The probability that all packets in Π_ℓ are absorbed in phase ϕ_ℓ is at least $1 - 1/(n \log 2n)$.*

Proof: From Lemma 2.18, any particular packet of Π_ℓ reaches its destination with probability at least $1 - 1/(n^2 \log 2n)$. Thus, a packet will not reach its destination with probability at most $1/(n^2 \log 2n)$. The number of packets in Π_ℓ is at most n (each node in the network injects at most one packet). By the union bound, the probability that one of these packets does not make it to the destination in phase ϕ_ℓ is at most $n \cdot 1/(n^2 \log n) = 1/(n \log 2n)$. Subsequently, all the packets make it to the destination with probability at least $1 - 1/(n \log 2n)$.
■

Corollary 2.20 *For $0 \le \ell \le m$, if $P_{\ell-1}$ holds, then P_ℓ holds with probability at least $1 - 1/(n \log 2n)$.*

We are now ready to show that properties P_ℓ hold with high probability:

Lemma 2.21 *For $0 \le \ell \le m$, P_ℓ holds with probability at least $1 - (\ell + 1)/(n \log 2n)$.*

Proof: Let $\overline{P}_i$ be the complementary event to P_i. Then $Pr[\overline{P}_i] = Pr[\overline{P}_i \cap P_{i-1}] + Pr[\overline{P}_i \cap \overline{P}_{i-1}]$.

$$\begin{aligned} Pr[\overline{P}_i \cap P_{i-1}] &= Pr[\overline{P}_i|P_{i-1}]Pr[P_{i-1}] \le Pr[\overline{P}_i|P_{i-1}], \\ Pr[\overline{P}_i \cap \overline{P}_{i-1}] &\le Pr[\overline{P}_{i-1}], \end{aligned}$$

so, using Corollary 2.20 we have that $Pr[\overline{P}_i] \le 1/(n \log 2n) + Pr[\overline{P}_{i-1}]$. Since $P_0 \le 1/(n \log 2n)$, the claim now follows by an easy induction. ■

[‡]Note that each deflection is treated as an independent event for reaching the destination node. We can do this because we have computed the $p/2$ lower bound for this probability for the worst possible scenario for each deflection. The consideration of the dependencies between deflections cannot possibly decrease the $p/2$ lower bound for each deflection.

From Lemma 2.21 and the fact that $m \leq \log n$, we obtain the following corollary:

Corollary 2.22 *P_m holds with probability at least $1 - 1/n$.*

Since $m \leq \log n$ and $\tau = O((C + D) \log n)$, Corollary 2.22 implies that with probability at least $1 - 1/n$, all packets are absorbed by time step $\tau \cdot (m + 1) \leq \kappa(C + D) \log^2 n$, for some constant $\kappa \approx 33$. Thus we have:

Theorem 2.23 *With probability at least $1 - 1/n$, the delivery time of Algorithm* Tree-Randomized *is bounded by $\kappa(C + D) \log^2 n$, for some constant $\kappa > 0$.*

3 Leveled Networks

In this section we give our results for leveled networks. We start in Section 3.1 where we describe the decomposition of the network into frames and groups. In the same section we also describe how to construct the dependency graphs and properties of them. We then continue with a description and analysis of the centralized in Section 3.2 and distributed algorithms in Section 3.3.

3.1 Preliminaries on Leveled Networks

We give some necessary preliminaries that will be used in our algorithms in leveled networks. Consider throughout this section a leveled network $G = (V, E)$ with $L + 1$ levels. We will need a Chernoff-type tail inequality.

Lemma 3.1 (Chernoff bound, [38, Exercise 4.1]) *Let $\{X_i\}_{i=1}^n$ be independent Bernoulli random variables, with $Pr[X_i = 1] = p_i$. Let $X = \sum_{i=1}^n X_i$, and set $\mu = E[X] = \sum_{i=1}^n p_i$. For any $\delta > 2e$, $Pr[X > \delta\mu] < 2^{-\delta\mu}$.*

We now introduce packet paths, oscillations, frames, and the dependencies between packets.

3.1.1 Paths and Oscillations in Leveled Networks

Similar to the trees, a packet path is a sequence of nodes $(v_i, v_{i+1}, \dots, v_k)$, where every pair of consecutive nodes in the path is an edge of the network. The preselected path is monotonic, it is a sequence of nodes from lower levels to higher levels. Similar to the trees, packets have current paths, initially the preselected path, which changes every time that the packets moves forward or gets deflected.

Suppose packet π has current path $(v_i, v_{i+1}, \dots, v_k)$. We say that π *oscillates* on edge $e = (v_i, v_{i+1})$ if it moves back and forth on e: if at time t, π appears in v_i, then at time $t + 1$, π appears in v_{i+1}, and at time $t + 2$ it is back in v_i, and so on. When a packet oscillates, the length of its current path increases and decreases by one each time. Oscillations are useful because they provide a way to "buffer" packets on edges instead of at nodes.

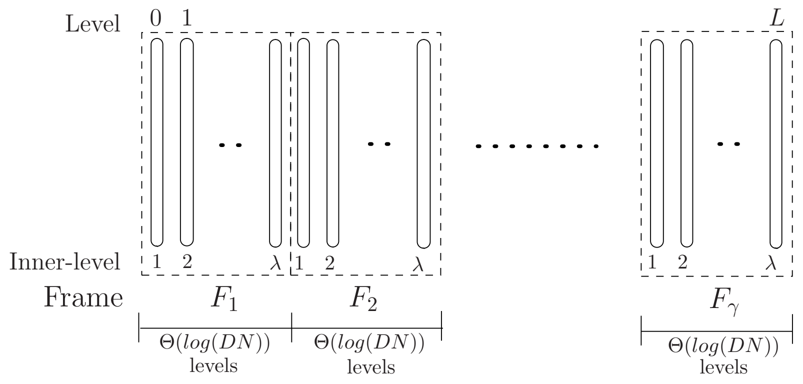

Figure 6: Network partition into frames

3.1.2 Frames

We partition the levels of the network into γ non overlapping *frames* $F_1, F_2, \ldots, F_\gamma$, where each frame is a collection of levels, each containing λ levels (except possibly for the last frame, which may contain fewer). Frame F_i, $1 \leq i < \gamma$, consists of the λ levels $(i-1)\lambda, \ldots, i\lambda - 1$. Frame F_γ consists of the levels $(\gamma - 1)\lambda, \ldots, L$. Note that $\gamma = \lceil (L+1)/\lambda \rceil$. We will pick $\lambda = 4\alpha \log(DN)$, where α is a parameter to be defined later; thus, the frames have logarithmic size. (We assume that $\log(DN)$ is an integer, if not we use $\lceil \log(DN) \rceil$.) The partition of the network into frames is depicted in Figure 6.

We refer to the levels that comprise frame F_i as the *inner-levels* of F_i, and we number them from 1 to λ. Thus, inner-level k of frame F_i corresponds to real level $(i-1)\lambda + (k-1)$, where $1 \leq k \leq \lambda$. The *odd inner-levels* are numbered $1, 3, \ldots, \lambda - 1$ (recall that λ is even). The inner level of an edge is the smaller of the inner-levels of the nodes it is incident with. Thus, corresponding to odd inner-levels are *odd inner-edges*, and similarly even inner-levels and even inner-edges.

3.1.3 Packet Sets and Dependency Graphs

Let Π be a set of packets in a many-to-one batch problem, with $|\Pi| = N$. We partition the set of packets Π into $s = 8\alpha e C$ sets, $\Pi_1, \Pi_2, \ldots, \Pi_s$. Each packet is placed into one of these sets uniformly at random. Thus, $\Pi = \bigcup_{i=1}^{s} \Pi_i$, and $\Pi_i \cap \Pi_j = \emptyset$ for $i \neq j$, so $|\Pi| = \sum_{i=1}^{s} |\Pi_i| = N$.

Consider the packets in Π_i, and two consecutive frames F_j and F_{j+1}. For each packet $\pi \in \Pi_i$ denote by q_π the sub-path of the preselected path that consists only of edges in F_j and F_{j+1}. We define the packet dependency graph $G_{(i,j)} = (V_{(i,j)}, E_{(i,j)})$ as follows. The nodes of $V_{(i,j)}$ correspond to the packets in Π_i, so $|V_{(i,j)}| = |\Pi_i|$. Let $\pi, \sigma \in \Pi_i$, then $(\pi, \sigma) \in E_{(i,j)}$ if and only if the paths q_π and q_σ share some edge in (F_j, F_{j+1}), i.e., if the paths collide.

The *degree* of a packet π in $G_{(i,j)}$, denoted $d_{(i,j)}(\pi)$, is the number of edges incident with π. The degree of $G_{(i,j)}$, denoted $d_{(i,j)}$, is the maximum degree of any packet in $V_{(i,j)}$. Let $d = \max_{\{i,j\}} d_{(i,j)}$, i.e., d is the maximum degree of any of the graphs $G_{(i,j)}$, for any i and j.

We show that d cannot be too big. In fact, a packet path collides with at most $2\lambda C$ other paths over two consecutive frames. Only approximately $2\lambda C/s = O(\lambda/\alpha)$ of these packets are in the same set, so we expect that $d = O(\lambda/\alpha)$. The next lemma formalizes this notion.

Lemma 3.2 *$d \leq \lambda/\alpha = 4\log(DN)$, with probability at least $1 - 1/DN$.*

Proof: Consider $d_{(i,j)}(\pi)$, for $\pi \in \Pi_i$. Note that $|q_\pi| \leq 2\lambda$. Let R denote the set of packets that collide with π on q_π, $|R| \leq |q_\pi|C \leq 2\lambda C$. Let $\sigma \in R$, then $Pr[\sigma \in \Pi_i] = \frac{1}{s}$. Therefore, $\mu = E[d_{(i,j)}(\pi)] = \sum_{\sigma\in R} Pr[\sigma \in \Pi_i] = \frac{|R|}{s} \leq \frac{\lambda}{4\alpha e}$. Since the events $\sigma \in \Pi_i$ are independent Bernoulli trials, we can apply Lemma 3.1 with $\delta = 2e + \frac{\lambda}{2\alpha\mu}$ to obtain

$$Pr[d_{(i,j)}(\pi) > \tfrac{\lambda}{\alpha}] < 2^{-(2e\mu+\frac{\lambda}{2\alpha})} \leq 2^{-\frac{\lambda}{2\alpha}}$$

d is the maximum degree over any node in any $G_{(i,j)}$. Since every packet has path length at most D, a packet appears as a node in at most D of the $G_{(i,j)}$'s. Thus, $\sum_{i,j} |V_{(i,j)}| \leq DN$. We now succesively apply the union bound to obtain the desired result:

$$\begin{aligned}
Pr[d_{(i,j)} > \tfrac{\lambda}{\alpha}] &= Pr[\max_{\pi\in V_{(i,j)}} d_{(i,j)}(\pi) > \tfrac{\lambda}{\alpha}] \\
&\leq |V_{(i,j)}| 2^{-\frac{\lambda}{2\alpha}} \\
Pr[d > \tfrac{\lambda}{\alpha}] &= Pr[\max_{i\in[1,s], j\in[1,\gamma]} d_{(i,j)} > \tfrac{\lambda}{\alpha}] \\
&\leq \sum_{i,j} |V_{(i,j)}| 2^{-\frac{\lambda}{2\alpha}} \leq DN\ 2^{-\frac{\lambda}{2\alpha}}.
\end{aligned}$$

Since $\lambda = 4\alpha\log(DN)$, the lemma follows. ■

3.1.4 Groups

We partition the network into *groups*, such that each group is a collection of γ' consecutive frames, where $\gamma' = 2\lceil D/\lambda \rceil$ (namely, the group consists of at most $2D + 2\lambda$ levels). We define two sets of groups (see Figure 7.) The first set of groups is $S_1 = \{g_1, g_2, \ldots, g_{k_1}\}$, where group g_i consists of frames $F_{(i-1)\gamma'+1}, \ldots, F_{i\gamma'}$. The group g_{k_1} consists of the rightmost frames in the network and may contain fewer than γ' frames. Note that the groups in S_1 do not share any levels. The second set of groups is $S_2 = \{h_1, h_2, \ldots, h_{k_2}\}$, where group h_i consists of frames $F_{(i-1/2)\gamma'+1}, \ldots, F_{(i+1/2)\gamma'}$. The group h_{k_2}, consists of the rightmost frames in the network and may contain fewer than γ' frames. Note that the groups in S_2 are shifted by $\gamma'/2$ frames with respect to the groups in S_1.

A packet belongs to a group if it lies entirely within the group. In S_1, a packet belongs to at most one group, since the packet length is at most D and the group size is approximately $2D + 2\lambda$. Similarly, in S_2 a packet belongs to at most one group. In the case where a packet belongs to two groups, one in S_1 and one in S_2, we assign the packet to the group in S_1; thus, a packet has always a unique group. A packet may belong to at most two groups, one group in S_1 and one group in S_2 (the packet is in the intersection of the group). In such a

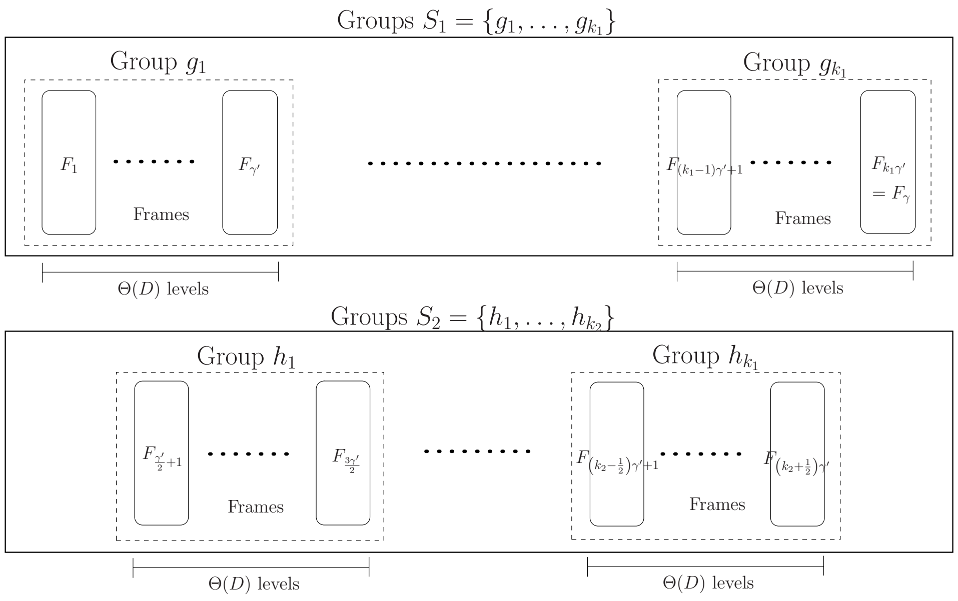

Figure 7: Network partition into groups of frames

case, we assign the packet to the group of S_1. A packet belongs to group set S_j if its group is in S_j. We denote by $\Pi(S_i)$ the packets that belong to group S_i, and by $\Pi(x, S_j)$ the set of packets that belong to group x of S_j. Further, $\Pi_i(x, S_j)$, denotes the subset of packets of Π_i belonging to group x of S_j.

3.2 A Centralized Algorithm for Leveled Networks

Here, we give Algorithm Leveled-Centralized (Algorithm 3). In the algorithm, we send the packets in two *sessions*, which we denote as $sess_1$ and $sess_2$. In the first session, $sess_1$, we send the packets $\Pi(S_1)$ (belonging to group set S_1) and in second session, $sess_2$, we send the packets $\Pi(S_2)$ (belonging to group set S_2). The second session begins after the first session ends.

In the main part of the Algorithm Leveled-Centralized we use the network decomposition into frames and the packet partition into sets as described in Section 3.1. The algorithm then invokes Algorithm Deliver-Group (Algorithm 4) which handles the delivery the packets in a group. Since in each group the packets are level-wise disjoint, the packets in one group can be sent simultaneously with all the packets in another group without any possibility of interfering. Thus, it suffices to describe the algorithm to deliver the packets in only one group; the rest of the groups are treated similarly. In the same group set, the packets in different groups can be delivered simultaneously, since the groups do not overlap.

We now describe Algorithm Deliver-Group. The input is the set of packets that belong to the group. In order to simplify the description of Deliver-Group, we focus on group $x = g_1$ of S_1. The algorithm for other groups is identical except for a change in the indices. Since we focus on the first group in the first session, we will simplify the notation by dropping the x and $\mathsf{S_j}$ dependence. Hence, in Algorithm Deliver-Group, Π will denote $\Pi(g_1, S_1)$, and Π_i will denote $\Pi_i(g_1, S_1)$. The session consists of m phases, each of duration τ time steps.

The basic idea of Deliver-Group, is that packets move on waves to their destinations. The waves move from left to right in the network, such that at each phase a wave moves one frame to the right. Each packet set Π_i has associated with it a particular wave which the packets follow until they are delivered to their destinations. At each phase, the packets move from frame to frame to the along their waves. We also have the notion of a "boat", which packets of independent sets follow in order to move along their waves. The detailed description of Deliver-Group follows below.

```
Algorithm: Leveled-Centralized
Input: A leveled network $G$ with $n$ nodes; A batch problem with packets $\Pi$ where
       $|\Pi| = N$; Packets have preselected paths with congestion $C$ and dilation $D$;
begin
    $\alpha = 2 + 1/(2\log(DN))$; $\lambda = 4\alpha\log(DN)$;
    $\gamma = \lceil (L+1)/\lambda \rceil$; $\gamma' = 2\lceil D/\lambda \rceil$;
    $s = 8\alpha e C$; $m = 2s + \gamma' - 1$;
    $\chi = \lambda/\alpha + 1$; $\tau = 2(\chi + \lambda - 1)$;
1   Partition $G$ into frames $F_1, \dots, F_\gamma$ of width $\lambda$;
2   Construct the group sets $S_1$ and $S_2$;
3   Partition $\Pi$ uniformly at random into sets $\Pi_1, \dots, \Pi_s$;
4   Construct all dependency graphs $G_{(i,j)}$;
5   Greedily color the nodes of each $G_{(i,j)}$ with at most $\chi$ colors;
6   Divide time into two consecutive sessions each consisting of $m \cdot \tau$ time steps;
7   for $j = 1, 2$ do
8       for session $sess_j$, and for each group $g \in S_j$ do
9           Deliver-Group($g, S_j$);
        end
    end
end
```

Algorithm 3: Leveled-Centralized

3.2.1 Waves

A *wave* ω is a pointer to a frame (see Figure 8). Initially the wave is NULL. The wave enters the network (points to frame F_1) at some phase ϕ_i. At each subsequent phase the wave points to the next higher frame, so in phase ϕ_{i+k}, it points to frame F_{k+1}. Eventually, ω points to the last frame $F_{\gamma'}$, after which it leaves the network and becomes NULL. There are s waves $\omega_1, \omega_2, \dots, \omega_s$ (as many waves as there are packet sets). Wave ω_i enters the network at phase ϕ_{2i-1}. Note that waves are spaced 2 frames apart, which will be useful for moving packets (see below). The last wave ω_s enters in phase ϕ_{2s-1} and after γ' phases, it has left

```
Algorithm: Deliver-Group($g_1, S_1$)
//Same algorithm applies for any group $g \in S_1$ and also for grouping $S_2$;
Input: Batch problem with packets $\Pi(g_1)$ in group $g_1$; Group $g_1$ consists of frames
       $F_1, \dots, F_{\gamma'}$;
begin
1    Divide time into phases $\phi_1, \dots, \phi_m$, each phase consisting of $\tau$ time steps;
2    Define waves $\omega_1, \dots, \omega_s$, where wave $\omega_i$ enters the network at phase $\phi_{2i-1}$;
3    for each packet set $\Pi_i$ do
4        Packets of set $\Pi_i$ follow wave $\omega_i$ as follows;
5        for each phase $\phi$ in which wave $\omega_i$ points to frame $F_j$ do
6            // Packets in $F_j$ will move to $F_{j+1}$;
7            Initially, only packets of $\Pi_i$ oscillate in $F_j$, and $F_{j+1}$ is empty;
8            Phase $\phi$ consists of time steps $t_1, t_2, \dots, t_\tau$;
9            Define boats $b_1, \dots, b_\chi$, where boat $b_k$ enters the network at time $t_{4k-3}$;
10           Packets of color $k$ follow boat $b_k$ to target inner-level $\ell_k = \lambda - (2k-1)$ in
             $F_{j+1}$, where they will oscillate until the next phase;
         end
     end
end
```

Algorithm 4: Deliver-Group

the network, so the number of phases is $m = 2s + \gamma' - 1$. We use the wave to also denote to the frame it points to.

The purpose of wave ω_i is to move the packets in set Π_i along with it, as it moves from lower to higher levels. Packet $\pi \in \Pi_i$ is injected when wave ω_i contains π's source. The packet is absorbed either when the wave contains its destination or its destination is one frame ahead of the wave.

At the beginning of each phase, packets appear inside their respective waves, and frames between waves are empty of packets; this property is essential for moving packets along their waves. Consider a phase ϕ during which wave ω_i points to frame F_j. At the beginning of ϕ, F_j contains only packets from Π_i, and F_{j+1} is empty of packets. By the end of phase ϕ, the packets in F_j will move from frame F_j to frame F_{j+1}. Thus, at the beginning of the next phase, all these packets are still in the wave ω_i, and frame F_j is empty (which allows packets of Π_{i+1} to move along wave ω_{i+1}). We continue by describing in detail how the packets of Π_i move from F_j to F_{j+1} during phase ϕ.

3.2.2 Initial and Target Levels

Consider again phase ϕ during which wave ω_i points to frame F_j, and the packets will move during the phase from F_j to F_{j+1}. Suppose that phase ϕ consists of time steps $t_1, t_2, \dots, t_\tau$. At the beginning of phase ϕ, the packets of Π_i that are already in wave ω_i are oscillating on odd inner-edges of F_j. Suppose $\pi \in \Pi_i$ is oscillating on odd inner-edge $e = (v_\ell, v_{\ell+1})$ of F_j,

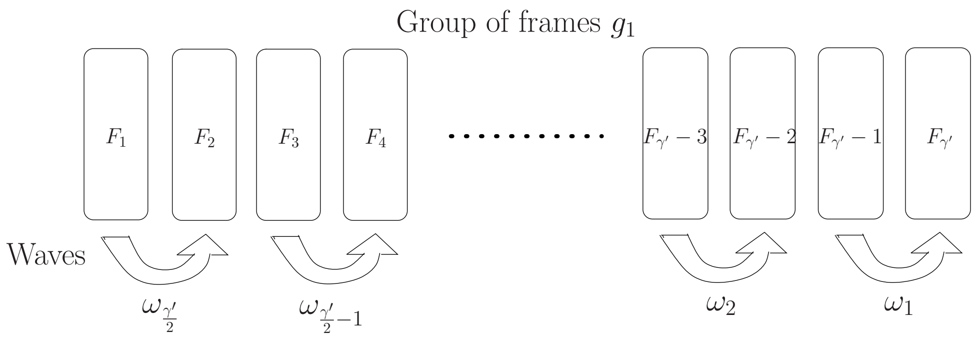

Figure 8: The waves

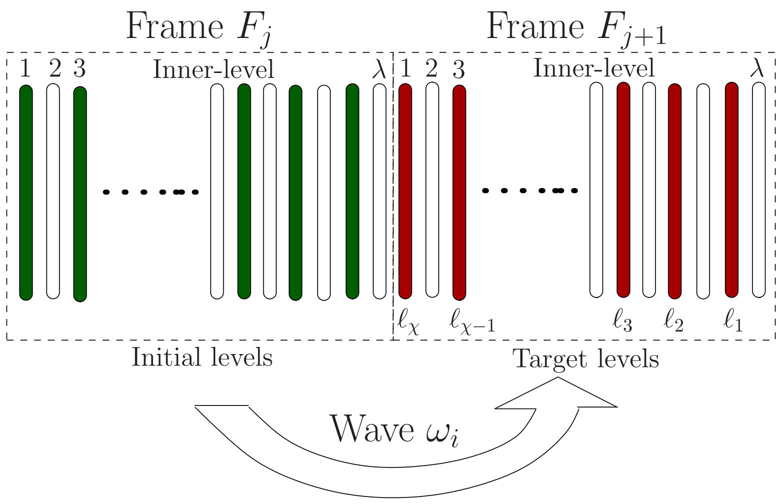

Figure 9: The initial and target levels

where the inner level of v_ℓ is ℓ (which is odd). The packet oscillates on e so that at odd time steps $t_1, t_3, \ldots$, packet π appears in v_ℓ. We say that π oscillates at inner-level ℓ, which is the *initial* inner-level of π in phase ϕ. (See Figure 9.)

Now suppose that the current path of π at its initial inner-level ℓ is a sub-path of its preselected path. During phase ϕ, packet π will follow its current path until it reaches a *target* inner-level ℓ' in F_{j+1}, where it will oscillate for the remainder of the phase. At its target level, π's current path will remain a sub-path of its preselected path. The target level will become the new initial level at the next phase, when the wave ω_i points to F_{j+1}.

We define $\chi_{(i,j)}$ different target inner-levels $\ell_1, \ell_2, \ldots, \ell_{\chi_{(i,j)}}$ in F_{j+1}, where ℓ_k is inner-level $\lambda-(2k-1)$ in F_{j+1}. (Note that target inner-levels are odd, because λ is even.) The parameter $\chi_{(i,j)}$ is the chromatic number of the dependency graph $G_{(i,j)}$. Since $d_{(i,j)} \leq d$, a trivial greedy polynomial time coloring algorithm using $d+1$ colors shows that $\chi_{(i,j)} \leq \chi = d+1$. Each packet in Π_i is thus assigned a color between 1 and $\chi_{(i,j)}$. Denote by $\Pi_i(k)$ the respective subset of Π_i with color k. Packets in $\Pi_i(k)$ have target level ℓ_k. By construction, the paths of packets of same color are *conflict-free*, i.e. don't share any edge, and thus can be sent together in a collision-free manner using "boats" (see below).

Note that in the above discussion we assume that $j < \gamma'$. If $j = \gamma'$ then all the target inner-levels are set to real level $2D - 1$, which are still in F_j. The fact that the last frame may extend beyond level $2D$ does not cause a problem because no packet will every need to move into that region, as it will be absorbed before that.

3.2.3 Boats

A *boat* b is a pointer to a level. We have $\chi_{(i,j)}$ boats $b_1, \ldots, b_{\chi_{(i,j)}}$. Initially, b_k is NULL. At time step t_{4k-3}, boat b_k points to the first inner-level of F_j (the boat enters the wave). At each subsequent step, the boat points to the next higher inner-level, so that at time step t_{4k-3+l} it points to inner-level $l + 1$. After the boat reaches the last inner-level of F_j it continues to the inner-levels of F_{j+1} until the boat reaches the target level ℓ_k of F_{j+1}, after which b_k becomes NULL again. Note that boats are spaced 4 levels away from each other, which will be important when an oscillating packet needs to be deflected (see below). When the context is clear, we use boat to refer to the inner-level it points to. Note that the last boat enters at $t_{4\chi_{(i,j)}-3}$, and takes $2\lambda - 2\chi_{(i,j)} + 1$ steps to leave the wave, so the number of time steps per phase is $\tau = 2(\lambda + \max_{i,j} \chi_{(i,j)} - 1) \leq 2(\lambda + \chi - 1)$.

The packets of $\Pi_i(k)$ will use boat b_k to move to their target level ℓ_k in F_{j+1}. Suppose $\pi \in \Pi_i(k)$ is oscillating with initial level ℓ at the beginning of phase ϕ. Packet π will continue to oscillate until its boat b_k is at inner-level ℓ, at which time packet π will "catch its boat" and move along with it. While on its boat b_k, π follows its current path until it reaches its target inner-level ℓ_k in F_{j+1}. If, during this trip, π passes through its target node it is absorbed; otherwise π reaches its target inner-level ℓ_k at which it will oscillate for the remainder of the phase. Note that b_k passes through odd inner-levels (in particular π's initial level) at odd time steps, so π is at its initial level when b_k passes through it.

Packet Injection. A packet $\pi \in \Pi_i(k)$ with source node in frame F_j, is injected into the network when its boat b_k passes through the source node. π then moves along with b_k, following its current path, until it reaches its target level ℓ_k. While a packet move along its boat it may collide with other packets; we now describe how to handle such collisions.

3.2.4 Packet Collisions

Suppose $\pi \in \Pi_i(k)$ is on its boat b_k, progressing along its current path to its target level ℓ_k. π cannot collide with another packet of $\Pi_i(k)$ because their current paths are conflict-free ($\Pi_i(k)$ is an independent set in $G_{(i,j)}$). Earlier boats $b_{k'}$ with $k' < k$ are ahead of b_k, so π cannot collide with packets in $\Pi_i(k')$. π can only collide with packets in $\Pi_i(k'')$ for $k'' > k$, which are oscillating in F_j. In such a collision, the oscillating packet is deflected (i.e., oscillating packets have lower priority than packets on boats). We show below that this does not disrupt the algorithm.

Suppose π deflects packet $\sigma \in \Pi_i(k'')$ which oscillates on edge $e = (v_\ell, v_{\ell+1})$ (ℓ is σ's inner-level in F_j). Packet π deflects σ at the (odd) time step t_k at which π passes through

ℓ. Assume that σ followed edge $e' = (v_{l-1}, v_l)$ to reach v_ℓ. We deflect σ along edge e' to inner-level $\ell - 1$, (so that at time step t_{k+1}, σ appears in v_l). Note that this is always possible because no other packet oscillating at v_ℓ arrived there using edge e', because the packets that are oscillating at v_ℓ all followed the same boat, and hence had edge disjoint paths. Note also that a packet oscillating on the first inner-level may be deflected into the previous frame F_{j-1} by an injected packet, but this causes no problem. Packet σ now follows edge e' to appear back in v_l at the (odd) time step t_{k+2}. This is possible because at time step t_{k+1} there is no boat passing through inner-level $\ell - 1$ (boat b_{k+1} is two levels away), and thus σ cannot be deflected further. When packet σ is back at inner-level ℓ, it continues to oscillate in ℓ. Therefore, σ is always at level ℓ at odd time steps, and thus it can move with boat $b_{k'}$, when it passes through ℓ. Clearly, deflected packets remain on their path.

3.2.5 Delivery Time

The only parameter that remains to be specified is α. This parameter determines the frame size λ which must be large enough to accommodate 2χ levels (at least χ odd target inner-levels) in F_{j+1}. Since $\chi \leq d + 1$, it suffices that $\lambda = 4\alpha \log(DN) \geq 2(d + 1)$. From Lemma 3.2, $d \leq \lambda/\alpha = 4 \log(DN)$ (with high probability), so $\alpha = 2 + 1/(2 \log(DN))$ will do. (We ignore the trivial case $D = N = 1$.)

The delivery time is at most $m \cdot \tau$ (m phases, each of duration τ). Since $m = O(s + \gamma') = O(C + D/\log(DN))$, and $\tau = O(\lambda + \chi) = O(\log(DN))$ with high probability (Lemma 3.2), we obtain:

Theorem 3.3 *The delivery time of Algorithm* Leveled-Centralized *is $O(C \log(DN) + D)$, with probability at least $1 - 1/DN$.*

3.3 A Distributed Algorithm for Leveled Networks

We show how to make the Algorithm Leveled-Centralized (see Section 3.2) distributed. The new algorithm will be called Leveled-Distributed. We will describe how to obtain the new distributed algorithm. We assume that all nodes know the parameters C, D, and N. Given C, D, N, every node can compute $\lambda, \gamma', s, m, \chi, \tau$. Note, that nodes do not need to know the paths of the other packets, they only need to know the path of the packet they inject.

The setup of the distributed algorithm is similar to the centralized algorithm: packets follow boats on waves to reach their destinations. The entire centralized algorithm would work verbatim if not for the coloring of the dependency graphs $G_{(i,j)}$, which is the main centralized computation (since nodes need to know all the packet paths). Thus, we need a distributed coloring algorithm, which will compute a coloring as the packets follow the waves. We introduce the notion of reverse simulation to accomplish this.

3.3.1 Reverse Simulation

Let $\chi = 2\lambda/\alpha$ (λ/α is an upper bound on d, with high probability, by Lemma 3.2). During phase ϕ, suppose that wave ω_i point to frame F_j. Packets of set Π_i follow wave ω_i. In F_j

and F_{j+1} we define the initial and target levels as in the centralized algorithm. Consider the set of packets $A \subseteq \Pi_i$ which are oscillating at their initial inner levels in frame F_j, at the beginning of the phase. These packets will move to F_{j+1}, where they will oscillate in their target levels.

As in the centralized algorithm, packets will use boats to move to their target levels. There are χ boats. In order to follow the boats in a collision-free manner, the packets need to be colored so that packets of same color have conflict-free paths. In order to obtain the colors, the packets will simulate a distributed coloring algorithm, which consists of several rounds. In the first round each packet chooses a color randomly and uniformly among χ colors. Packets can then be divided into two disjoint sets: those that obtained a valid color (one that is different from the color of each of their neighbors in $G_{(i,j)}$); those that obtained an invalid color (one that coincides with the color of at least one neighbor). Packets now attempt to reach their target inner-levels. Packets assigned invalid colors will detect this when they collide with non-oscillating packets, and will attempt to correct this in the next round. This process continues until all packets have obtained valid colors. We now give the details.

A phase is divided into ξ rounds $r_1, r_2, \ldots, r_\xi$, and each round consists of 2τ time steps, which is twice the duration of a phase in the centralized algorithm; this is because packets will need to attempt to reach their target level, and return to their initial level in each round. Each round has χ boats and target levels (similar to the centralized algorithm). At the beginning of round r_1, each packet in A chooses a color uniformly and randomly among χ colors. Let A_1 be the set of packets with a valid color, and A'_1 the packets with an invalid color. Note that $A = A_1 \cup A'_1$.

During round r_1, *all* packets in A will follow their respective boats. The packets in A_1 will not be deflected, and they follow their respective boats to successfully reach their target levels where they will oscillate for the rest of the round. Some packets, $A''_1 \subseteq A'_1$, will collide with non-oscillating packets as they follow their boats. Such packets can mark themselves as members of A'_1. These packets need to choose new colors and try again. At the end of round r_1, *all* packets in A return to their initial level (see below). In round r_2, packets in set A''_1 choose a new color, and a subset $A'_2 \subseteq A''_1$ will still have an invalid color. A subset $A''_2 \subseteq A'_2$ will collide with non-oscillating packets, and will need to choose new colors in the next round. Continuing in this way, in round k, the packets in A''_{k-1} choose new colors, and those in $A'_k \subseteq A''_{k-1}$ still do not have a valid color. Of these packets, A''_k will collide with non-oscillating packets. We will show A'_ξ is empty w.h.p, i.e., all packets have a valid color by the last round. Thus, in the last round, all the packets reach their target inner-levels, where they will oscillate till the next phase. We give the details below.

We define 4 levels of priority, $0, 1, 2, 3$. When two or more packets collide, the packet with highest priority always wins, and ties are broken randomly. A packet which successfully reaches its target level in round k (without being deflected by non-oscillating packets) keeps its color in all subsequent rounds and attains priority 3 for the remainder of the phase, whenever it is not oscillating. An oscillating packet has priority 1. A packet that chooses a new color in a round attains priority 2 for the round. If, during the round, it collides with any priority 2 or 3 packet, it immediately attains priority 0 for the remainder of the round, and will select a new color in the next round. Such priority 0 packets do not "distract" other forward going packets, and they follow arbitrary paths, due to deflections, for the remainder of the round.

At the end of a round, all packets in A (with valid or invalid coloring) need to appear back at their initial levels. Let t be the time step that the last boat in the round leaves the network. After time t, all packets follow, in reverse, the path that they followed from the beginning of the round. Thus, by the end of the round, they appear at their initial level where they oscillate until the next round. The path reversal is accommodated by having the nodes store all their computations from the beginning of the round up to time t. After time t, the nodes simply do the reverse computations. (This is why we need the round to be twice as long as τ.)

3.3.2 Packet Injections

So far we considered only the oscillating packets in Π_i, that already appear in F_j at the beginning of phase ϕ. We also need to consider the set of packets $B \subseteq \Pi_i$ that will be injected in F_j during ϕ. Packets of B can be further partitioned into two sets: B_1, which are the packets of B whose source are at odd inner-levels of F_j, and B_2, which have sources at even inner-levels of F_j. Packets of B_1 and B_2 are treated separately so that they can not interfere with each other.

We divide phase ϕ into three sub-phases ϕ_A, ϕ_{B_1}, and ϕ_{B_2} in which we send the packets of the respective sets A, B_1 and B_2 to F_{j+1}. Each sub-phase consists of ξ rounds. We also divide the frame F_{j+1} into three disjoint regions F_A, F_{B_1}, and F_{B_2}, each consisting of 2χ inner-levels and containing χ target levels. Region F_A occupies the upper one-third (right) inner-levels of F_{j+1}, F_{B_1} the middle one-third inner-levels, and F_{B_2} at the lower (left) one-third inner-levels. Packets of set A, B_1 and B_2, have their target levels in F_A, F_{B_1} and F_{B_2}, respectively.

During phase ϕ_A the packets of set A will move to region F_A, using the algorithm we described in Section 3.3.1. During ϕ_{B_1}, the packets of B_1 are injected into the network, and then they move to their target levels in region F_{B_1} using the reverse simulation technique that was used for packets in set A. The initial levels of the packets in B_1 are the inner-levels of their sources, and the packets are injected at the beginning of phase ϕ_{B_1}. Since a node injects at most 1 packet, the packets are guaranteed to be able to oscillate on their initial inner-levels during the reverse simulation. At the beginning of phase ϕ_{B_2}, the packets of set B_2 are injected into the network. Those packets will move to their target levels in region F_{B_2} during phase ϕ_{B_2} using the reverse simulation technique that was used for packets in set A. Those packets will also oscillate on their initial inner-levels, which are even (as opposed to packets in A and B_1 which have odd initial inner-levels). In order to handle the even levels, during this phase the boats enter the frame F_j from inner-level 2.

3.3.3 Delivery Time

First we determine an acceptable value of parameter α. A frame needs 2χ inner-levels for each of F_A, F_{B_1}, and F_{B_2}, so $\lambda \geq 6 \cdot \chi$ Since $\chi = 2\lambda/\alpha$, we obtain $\alpha \geq 12$, so $\alpha = 12$ will do.

We now choose the parameter ξ so as to ensure that the packet delivery is successful with high probability. Assume that $\chi \geq 2d$. Then, in a round, a packet picks a valid color with probability at least $\frac{1}{2}$. We denote a phase in which a packet eventually picks a valid color in one of the rounds as a successful phase for the packet. If $\xi = 2\log(LN)$, then a particular phase will be unsuccessful for a particular packet with probability $2^{-2\log(DN)} = 1/(DN)^2$. The distributed algorithm will be successful if every phase that a packet participates in is successful. A packet participates in at most D/λ phases (as its path length is at most D). A packet is unsuccessful if one of its phases is successful, therefore, by the union bound, a packet will be unsuccessful with probability at most $D/\lambda(DN)^2$. The distributed algorithm will be unsuccessful if one of the packets is unsuccessful, so applying the union bound again, the probability of failure is at most $DN/\lambda(DN)^2 < 1/DN$. Finally, by the union bound and Lemma 3.2, the probability that $\chi < 2d$ or some packet fails in some phase is at most $2/DN$.

There are m phases, and each phase has 3ξ rounds (since each sub-phase to send A, B_1, B_2 consists of ξ rounds), each of length 2τ time steps. Therefore, the delivery time is $O(m\xi\tau)$. Since $\tau = O(\log(DN))$, $m = O(C + D/\log(DN))$, and $\xi = O(\log(DN))$, we obtain:

Theorem 3.4 *The delivery time of Algorithm* Leveled-Distributed *is* $O(C\log^2(DN) + D\log(DN))$*, with probability* $1 - O(1/DN)$.

4 Discussion

We studied many-to-one batch problems with preselected paths on trees and leveled networks. We gave two algorithms for trees. The deterministic algorithm which is appropriate for trees whose degree is bounded by a constant and achieves delivery time $O(\mathcal{T}^* \cdot \lg n)$. The randomized algorithm is appropriate for arbitrary trees and achieves delivery time $O(\mathcal{T}^* \cdot \lg^2 n)$ with high probability. In both cases, T^* refers to the minimum possible routing time achievable by *any* routing algorithm (with or without buffers) for the given sources, destinations, and preselected paths. For leveled networks, we obtained similar results; we gave an $O(\mathcal{T}^* \cdot \log(n))$ centralized algorithm, which can be made distributed at a cost of an extra logarithm. Thus, bufferless packet switching on trees and leveled networks is within one logarithmic factor from optimal. Making the model stronger, i.e. considering distributed or randomized algorithms, we have an additional logarithmic factor.

Our algorithms show that bufferless packet switching can be efficient, and can achieve performance almost as good as in store-and-forward algorithms. Further, we show that the problem of efficient packet switching on trees and leveled bufferless networks can be reduced to the problem of starting with good preselected paths on them. Finding good paths is a classic research problem. We thus give a framework for systematic analysis of bufferless algorithms, where the routing task (finding paths) is separated from the scheduling task. In the other literature, the bufferless algorithms are studied in an ad hoc manner, giving good bounds only for worst cases of special batch problems. Our results are general and consider a wide range of batch problems.

Natural directions for future investigation are to remove the requirements that the nodes need to know C, D, N in the distributed algorithms. Further, an interesting problem is to determine whether there exist optimal algorithm for trees and leveled networks or for arbitrary networks without the extra logarithmic factors.

References

[1] A. S. Acampora and S. I. A. Shah. Multihop lightwave networks: a comparison of store-and-forward and hot-potato routing. In *Proc. IEEE INFOCOM*, pages 10–19, 1991.

[2] N. Alon, F.R.K. Chung, and R.L.Graham. Routing permutations on graphs via matching. *SIAM Journal on Discrete Mathematics*, 7(3):513–530, 1994.

[3] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Direct routing on trees. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 98)*, pages 342–349, 1998.

[4] J. Aspens, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. Online load balancing with applications to machine scheduling and virtual circuit routing. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 623–631, 1993.

[5] B. Awerbuch and Y. Azar. Local optimization of global objectives: competitive distributed deadlock resolution and resource allocation. In *Proceedings of 35th Annual Symposium on Foundations of Computer Science*, pages 240–249, Santa Fe, New Mexico, 1994.

[6] A. Bar-Noy, P. Raghavan, B. Schieber, and H. Tamaki. Fast deflection routing for packets and worms. In *Proceedings of the Twelth Annual ACM Symposium on Principles of Distributed Computing*, pages 75–86, Ithaca, New York, USA, August 1993.

[7] P. Baran. On distributed communications networks. *IEEE Transactions on Communications*, pages 1–9, 1964.

[8] C. Bartzis, I. Caragiannis, C. Kaklamanis, and I. Vergados. Experimental evaluation of hot-potato routing algorithms on 2-dimensional processor arrays. In *EUROPAR: Parallel Processing, 6th International EURO-PAR Conference*, pages 877–881. LNCS, 2000.

[9] I. Ben-Aroya, T. Eilam, and A. Schuster. Greedy hot-potato routing on the two-dimensional mesh. *Distributed Computing*, 9(1):3–19, 1995.

[10] I. Ben-Aroya, I. Newman, and A. Schuster. Randomized single-target hot-potato routing. *Journal of Algorithms*, 23(1):101–120, April 1997.

[11] A. Ben-Dor, S. Halevi, and A. Schuster. Potential function analysis of greedy hot-potato routing. *Theory of Computing Systems*, 31(1):41–61, January/February 1998.

[12] P. Berenbrink and C. Scheideler. Locally efficient on-line strategies for routing packets along fixed paths. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 112–121, N.Y., January 17–19 1999. ACM-SIAM.

[13] S. N. Bhatt, G. Bilardi, G. Pucci, A. G. Ranade, A. L. Rosenberg, and E. J. Schwabe. On bufferless routing of variable-length message in leveled networks. *IEEE Trans. Comput.*, 45:714–729, 1996.

[14] A. Borodin and J. E. Hopcroft. Routing, merging, and sorting on parallel models of computation. *Journal of Computer and System Sciences*, 30(1):130–145, February 1985.

[15] A. Borodin, Y. Rabani, and B. Schieber. Deterministic many-to-many hot potato routing. *IEEE Transactions on Parallel and Distributed Systems*, 8(6):587–596, June 1997.

[16] J. T. Brassil and R. L. Cruz. Bounds on maximum delay in networks with deflection routing. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):724–732, July 1995.

[17] A. Broder and E. Upfal. Dynamic deflection routing on arrays. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, pages 348–358, May 1996.

[18] C. Busch. $\tilde{\mathrm{O}}$(Congestion + Dilation) hot-potato routing on leveled networks. In *Proceedings of the Fourteenth ACM Symposium on Parallel Algorithms and Architectures*, pages 20–29, August 2002.

[19] C. Busch, M. Herlihy, and R. Wattenhofer. Hard-potato routing. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, pages 278–285, May 2000.

[20] C. Busch, M. Herlihy, and R. Wattenhofer. Routing without flow control. In *Proceedings of the Thirteenth ACM Symposium on Parallel Algorithms and Architectures*, pages 11–20, July 2001.

[21] C. Busch, M. Magdon-Ismail, and M. Mavronicolas. Universal bufferless routing. In *Proceedings of the 2nd Workshop on Approximation and Online Algorithms (WAOA)*, volume LNCS 3351, pages 239–252, September 2004.

[22] C. Busch, M. Magdon-Ismail, M. Mavronicolas, and P. Spirakis. Direct routing: Algorithms and Complexity. In *Proceedings of the 12th Annual European Symposium on Algorithms (ESA)*, volume LNCS 3221, pages 134–145, September 2004. Also to appear in Algorithmica, special issue with invited papers from ESA 2004.

[23] U. Feige. Nonmonotonic phenomena in packet routing. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing*, pages 583–591, New York, May 1–4 1999. ACM Press.

[24] U. Feige and P. Raghavan. Exact analysis of hot-potato routing. In IEEE, editor, *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 553–562, Pittsburgh, PN, October 1992.

[25] A. G. Greenberg and J. Goodman. Sharp approximate models of deflection routing. *IEEE Transactions on Communications*, 41(1):210–223, January 1993.

[26] B. Hajek. Bounds on evacutation time for deflection routing. *Distributed Computing*, 1:1–6, 1991.

[27] W. D. Hillis. *The Connection Machine*. MIT press, 1985.

[28] C. Kaklamanis, D. Krizanc, and S. Rao. Hot-potato routing on processor arrays. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 273–282, Velen, Germany, June 30–July 2, 1993.

[29] M. Kaufmann, H. Lauer, and H. Schroder. Fast deterministic hot-potato routing on meshes. In Springer-Verlag, editor, *Proceedings of the 5th International Symposium on Algorithms and Computation (ISAAC), LNCS*, volume 834, pages 333–341, 1994.

[30] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays - Trees - Hypercubes*. Morgan Kaufmann, San Mateo, 1992.

[31] F. T. Leighton, B. M. Maggs, and S. B. Rao. Packet routing and job-scheduling in $O(congestion + dilation)$ steps. *Combinatorica*, 14:167–186, 1994.

[32] F. T. Leighton, Bruce M. Maggs, Abhiram G. Ranade, and Satish B. Rao. Randomized routing and sorting on fixed-connection networks. *J. Algorithms*, 17(1):157–205, 1994.

[33] T. Leighton, B. Maggs, and A. W. Richa. Fast algorithms for finding O(congestion + dilation) packet routing schedules. *Combinatorica*, 19:375–401, 1999.

[34] N. F. Maxemchuk. Comparison of deflection and store and forward techniuques in the Manhattan street and shuffle exchange networks. In *Proc. IEEE INFOCOM*, pages 800–809, 1989.

[35] F. Meyer auf der Heide and C. Scheideler. Routing with bounded buffers and hot-potato routing in vertex-symmetric networks. In Paul G. Spirakis, editor, *Proceedings of the Third Annual European Symposium on Algorithms*, volume 979 of *LNCS*, pages 341–354, Corfu, Greece, 25–27 September 1995.

[36] F. Meyer auf der Heide and B. Vöcking. Shortest-path routing in arbitrary networks. *Journal of Algorithms*, 31(1):105–131, April 1999.

[37] M. Mitzenmacher. Constant time per edge is optimal on rooted tree networks. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 162–169, 1996.

[38] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[39] I. Newman and A. Schuster. Hot-potato algorithms for permutation routing. *IEEE Transactions on Parallel and Distributed Systems*, 6(11):1168–1176, November 1995.

[40] R. Ostrovsky and Y. Rabani. Universal $O(\text{congestion}+\text{dilation}+\log^{1+\varepsilon} N)$ local control packet switching algorithms. In *Proceedings of the 29th Annual ACM Symposium on the Theory of Computing*, pages 644–653, New York, May 1997.

[41] G. E. Pantziou, A. Roberts, and A. Symvonis. Many-to-many routing on trees via matchings. *Theoretical Computer Science*, 185(2):347–377, 1997.

[42] R. Prager. An algorithm for routing in hypercube networks. Master's thesis, University of Toronto, Computer Science Department, 1986.

[43] Y. Rabani and É. Tardos. Distributed packet switching in arbitrary networks. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, pages 366–375, Philadelphia, Pennsylvania, 22–24 May 1996.

[44] P. Raghavan and C. D. Thompson. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7:365–374, 1987.

[45] R. Ramaswami and K. N. Sivarajan. *Optical Networks, a Practical Perspective.* Morgan Kaufmann, 1998.

[46] A. Roberts, A. Symvonis, and D. R. Wood. Lower bounds for hot-potato permutation routing on trees. In M. Flammini, E. Nardelli, G. Proietti, and P. Spirakis, editors, *Proceedings of the 7th Int. Coll. Structural Information and Communication Complexity, SIROCCO*, pages 281–295. Carleton Scientific, 20–22 June 2000.

[47] C. L. Seitz. The caltech mosaic C: An experimental, fine-grain multicomputer. In *Proceedings of the 4th Symp. on Parallel Algorithms and Architectures*, June 1992. Keynote Speech.

[48] B. Smith. Architecture and applications of the HEP multiprocessor computer system. In *Proceedings of the 4th Symp. Real Time Signal Processing IV*, pages 241–248. SPIE, 1981.

[49] P. Spirakis and V. Triantafillou. Pure greedy hot-potato routing in the 2-D mesh with random destinations. *Parallel Processing Letters*, 7(3):249–258, September 1997.

[50] A. Srinivasan and C-P. Teo. A constant factor approximation algorithm for packet routing, and balancing local vs. global criteria. In *Proceedings of the ACM Symposium on the Theory of Computing (STOC)*, pages 636–643, 1997.

[51] A. Symvonis. Routing on trees. *Information Processing Letters*, 57(4):215–223, 1996.

[52] T. Szymanski. An analysis of "hot potato" routing in a fiber optic packet switched hypercube. In *Proc. IEEE INFOCOM*, pages 918–925, 1990.

[53] L. Zhang. Optimal bounds for matching routing on trees. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 445–453, 1997.

[54] Z. Zhang and A. S. Acampora. Performance analysis of multihop lightwave networks with hot potato routing and distance age priorities. In *Proc. IEEE INFOCOM*, pages 1012–1021, 1991.

A Algorithm Find-Short-Node

Algorithm Find-Short-Node (Algorithm 5) computes a short node for a tree.

Lemma A.1 *For a tree T with n nodes, Algorithm* Find-Short-Node *finds a short node of T in $O(n)$ time.*

Proof: Suppose that the node X in the algorithm is not short. Then in T^r there is a subtree T' of X with size M, where $M > n/2$. Let X' be the node of T' adjacent to X. With respect to X', the subtree containing the old node X has $n - M < n/2$ nodes and therefore if X' is not short, it cannot be due to its subtree containing X. Therefore the procedure will traverse a sequence of nodes without repeating a node. Since there are only n nodes, this process must stop, either at a short node in which case we are done, or when there is no where left to go (i.e., at a "leaf" node). The second case is impossible because the node from which it came will define a subtree with $n-1 \geq n/2$ nodes, which contradicts the fact that it must have $< n/2$ nodes.

Note that X' is a child of X in the rooted tree T^r. Thus, the sizes of the subtrees computed in the pre-order traversal of T^r, give the correct size for T'. The pre-order traversal on T^r requires $O(n)$ time to compute the sizes of all the subtrees in T^r. Thus, the entire procedure to find a short node takes $O(n)$ time. ■

```
Algorithm: Find-Short-Node(tree $T$)
Input: A tree $T$ with $n$ nodes $v_1, \ldots, v_n$.
Output: A short node of $T$.
begin
1   $r \leftarrow$ any arbitrary node of $T$;
2   Let $T^r$ be the rooted tree with root $r$. Using a standard pre-order traversal on
    $T^r$, compute for every node $v_i$, the number of nodes in the subtree of $T^r$ which is
    rooted at $v_i$;
3   $X \leftarrow r$;
4   while $X$ is not short do
5       Let $T'$ be a subtree of $X$ in $T$ which contains more than $n/2$ nodes;
6       Let $X'$ be the node of $T'$ which is adjacent to $X$ (i.e., the "root" node of $T'$);
7       $X \leftarrow X'$;
    end
8   return $X$;
end
```
Algorithm 5: Find-Short-Node