Optimal Local Buffer Management for Information Gathering with Adversarial Traffic *

[Regular Paper]

Stefan Dobrev †
Inst. of Mathematics
Slovak Academy of Sciences
Bratislava, Slovakia
Stefan.Dobrev@savba.sk

Manuel Lafond ‡
School of Eng. and Comp, Sci.
Université d'Ottawa
Ottawa, Canada
lafonman@iro.umontreal.ca

Lata Narayanan §
Dept. Comp. Sci. & Soft.Engg.
Concordia University
Montréal, QC, Canada
lata@cs.concordia.ca

Jaroslav Opatrny
Dept. Comp. Sci. & Soft.Engg.
Concordia University
Montréal, QC, Canada
opatrny@cs.concordia.ca

ABSTRACT

We consider a well studied problem in adversarial queuing theory, namely routing on directed paths and trees with a single destination, with rate-limited adversarial traffic. In particular, we focus on local buffer management algorithms that ensure no packet loss, while minimizing the size of the required buffers.

While a centralized algorithm for the problem that uses constant-sized buffers has been recently shown [1], there is no known *local* algorithm that achieves a sub-linear buffer size. In fact, greedy algorithms, which have been extensively studied in this setting, have been shown to require buffers of size $\Theta(n)$. In this paper we show tight bounds for the maximum buffer size needed by local algorithms for information gathering on directed paths and trees.

We show three main results:

- a lower bound of $O(\log n/l)$ for all ℓ-local algorithms on both directed and undirected paths.
- a surprisingly simple 1-local algorithm for directed paths that uses buffers of size $O(\log n)$.
- a natural extension of this algorithm to directed trees, achieving the same asymptotic bound. Here, our algorithm is 2-local in order to avoid the simultaneous sending of packets by siblings to a common successor.

Our $\Omega(\log n)$ lower bound is significantly lower than the $\Omega(n)$ lower bound for greedy algorithms; what is perhaps the most surprising is that there is a matching upper bound. The algorithm that achieves it can be summarized in two lines: If the size of your buffer is odd, forward a message if your successor's buffer size is equal or lower. If your buffer size is even, forward a message only if your successor's buffer size is strictly lower. In trees, an arbitration between siblings is needed, and is equally simple: The sibling with the fullest buffer has priority, if there are several of equal buffer size; choose one of them arbitrarily.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Geometrical problems and computations*

Keywords

Buffers; Buffer size; Routing; Trees; Directed paths; Information gathering; Local algorithms; Adversarial traffic

*(Produces the permission block, and copyright information). For use with SIG-ALTERNATE.CLS. Supported by ACM.

†Supported by VEGA grant

‡Supported in part by NSERC grant.

§Supported in part by NSERC grant.

SPAA'17, July 24–26, 2017, Washinghton, DC, USA.

DOI: 10.1145/1235

1. INTRODUCTION

Buffer or queue management in packet-switched networks has been an extensive subject of study for decades. Early theoretical work in the area studied *static* routing problems; the source-destination pairs corresponding to a finite set of packets is given as input to the network, and the goal is to route packets from their sources to their respective destinations, while minimizing the worst-case arrival time as well as the maximum size of buffer needed. In the case when multiple routes use the same link, a node with multiple incoming links stores incoming packets in a buffer, and uses a *buffer management* or *scheduling* policy that dictates which packet, if any, should be forwarded along each output port in each step. Well-known examples of scheduling policies include First-In-First-Out (FIFO), Last-in-First-Out (LIFO), Furthest-to-Go (FTG), Nearest-to-Go (NTG), etc. The policy used for buffer management has an impact on many crucial quality-of-service parameters for networks such as worst or average-case delay, jitter, packet loss, and the ability to provide differentiated service.

More recently, buffer management has been studied in the context of *dynamic* routing, where packets are continuously injected into the network. One way to analyze the performance of a buffer management strategy would be to use real-world data or stochastic models for network traffic. In a seminal paper, Borodin *et al* [2] introduced an *adversarial* model for traffic to analyze the *worst-case performance* of a scheduling strategy for dynamic routing. In this model, time proceeds in discrete steps. Given a network, in every step, an adversary injects packets at a certain set of nodes, and specifies, for each packet, a path to a destination, upon reaching which it will be *consumed*. The scheduling policy now chooses at most one packet to forward over each link of the network. Clearly, the network would be overwhelmed if the adversary generates more packets than can be sustained by the bandwidth in the network. Therefore, the adversary is assumed to be *rate-constrained*.

A key question of interest is whether a given scheduling policy is *stable* for a given network; that is, if the sizes of the buffers remain bounded. One class of scheduling strategies that has been extensively studied is the so-called *greedy* or *work-conserving* policies, wherein a packet is always forwarded along an edge e if there are packets waiting to use e. It has been shown that there are work-conserving policies that are stable; however, the size of the buffers, while independent of the length of the input stream, can be polynomial in the size of the network in the worst case. Even when the network is a path, the worst-case size of buffers for the greedy algorithm is $\Omega(n)$ [3].

In this paper, we study a class of routing problems called *information gathering* or *convergecast*, where the network has a special node called the *sink node*, and all packets generated in the network are destined for the sink. Such a communication pattern has been widely studied, particularly in the case of sensor networks, where sensor nodes are tasked with collecting data about their local environments and forwarding it to a sink node for processing. We are interested in *local* scheduling policies: every node must make its decision based on the contents of its own buffer, and knowledge of the buffer sizes of nodes in its ℓ-neighborhood. The goal of our work is to find upper and lower bounds on the buffer size required to achieve convergecast on trees with *no packet loss*, while using a *local* scheduling policy.

1.1 Related work

Adversarial queuing theory was introduced in [2] as a new approach to the study of queueing networks in general, and in particular, to the performance of scheduling algorithms in the context of dynamic or continuous packet routing in a network. Rather than considering the traffic to be stochastically generated, the authors proposed a fixed-rate adversary to generate the input; that is, the nodes where new packets are injected into the network, together with paths specified to their respective destinations. The main question considered in [2] was the *stability* of a queuing discipline for a particular network, *viz.* given a network G and a scheduling policy $\mathcal{S}$, is there a constant M (which can depend on the size of the network but is independent of the length of the input stream) so that for any adversarially generated input stream, the size of all buffers in the network remain bounded by M? Related questions of interest that were posed were the existence of universally stable policies, i.e. stable for all networks, and universally stable networks, i.e. stable for all policies of a given class. It was shown in [2] that every greedy queueing discipline is stable for rate 1 adversaries on any DAG, and that Furthest-to-Go is stable for rate 1 adversaries in a uni-directional ring. This result was extended in [4] to show that every greedy queueing discipline is stable for rate 1 adversaries in a unidirectional ring. Andrews et al [4] also showed that certain scheduling policies, such as Farthest-to-Go (FTG), Nearest-to-Source (NTS), Longest-in-System (LIS), and Shortest-in-System (SIS) are *universally stable*, that is, stable against rate < 1 adversaries in every network, while common policies such as FIFO, LIFO, and NTG, and Farthest-from-source (FTS) are not universally stable, and are not stable even on arrays and hypercubes. However, the aforementioned policies were shown to require queues and delays of size exponential in the size of the network in the worst case. They also showed that universal stability of a network can be decided in polynomial time. Finally, they give a local distributed and randomized scheduling policy that uses polynomial size buffers in the worst case. The problem has since been extensively studied, see for example, [5, 6, 7, 8, 9, 10, 11, 12].

Aeillo et al [13] proposed the related *Competitive Network Throughput* model in which the buffer size at every node is fixed to a constant B in advance, and the goal is to minimize the number of dropped packets; more specifically to maximize the competitive throughput, *viz.* the number of packets that are delivered to their destinations, relative to the best centralized offline algorithm that has access to the entire packet arrival sequence in advance. They show that all greedy protocols have bounded competitive ratio on DAGs. NTG, FTS, and LIS have competitive ratios that are bounded for all networks, while FTG, NTS, SIS have an unbounded competitive ratio on cycles. For the line network, it has been shown that if $B = 1$, any online deterministic algorithm is $\Omega(n)$ competitive while for $B > 1$, competitive ratio of $O(\sqrt{n})$ can be achieved. A lot of research was subsequently done using this framework, see for example [14, 15, 16, 3, 17, 18, 19].

In information gathering on a line, all packets are destined for the last node on the line. For this routing problem, all greedy protocols are identical from the point of view of throughput or packet loss. A lower bound of $\Omega(\sqrt{n})$ on the competitive ratio of the greedy protocol was given in [13]. Rosen and Scalosub [3] give tight bounds on the com-

petitive ratio of the greedy algorithm as a function of the injection rate of the adversary and the buffer size B. Their results imply that the greedy policy requires $\Theta(n)$ sized buffers to assure no packet loss. Azar and Zachut [17] study packet routing and information gathering in lines, rings, and trees, and show an $O(\log n)$-throughput-competitive deterministic algorithm. For lines and rings, they give an $O(\log^2 n)$-throughput-competitive randomized algorithm for multi-destination routing. Even and Medina [18] give an improved $O(\log n)$-competitive (randomized and centralized) algorithm for the multi-destination routing problem on a line, and extend the result to an $O(\log^{O(1)} n)$-competitive algorithm for directed grids [19, 20].

The papers closest to our work are [1] and [21]. Boaz-Shamir and Miller [1] study the same problem as the current paper. They consider a more general injection model with injection rate ρ (equal to link capacities) and burstiness bounded by σ. In this model they give a centralized algorithm that achieves information gathering without packet loss using buffers of size $\sigma + 2\rho$[1] and provide a matching lower bound to show that this is indeed the optimal buffer size. The algorithm, called *Forward-If-Empty (FIE)*, is unavoidably centralized, relying on simultaneously forwarding long *trains* of packets. As a way to motivate their centralized algorithm, they also analyze several local algorithms and for each of them show that in the worst case the buffer sizes are either unbounded, or at least $\Omega(n)$.

Kothapalli and Scheideler [21] study the competitive ratio of the buffer size achieved by algorithms for the problem of information gathering on an *undirected* path. Their adversarial model is significantly different and much stronger than ours: their adversary can not only choose the site of packet injection, but can also decide which edges are active [2]. They show a lower bound of $\Omega(\log n)$ on buffer sizes, as well as an algorithm which asymptotically matches this bound. Their algorithm forward packets in both directions, both toward and away from the sink, and therefore does not work on the directed path. In a follow-up paper [22], the authors show that with their adversary, any deterministic algorithm requires $\Omega(n)$-sized buffers in spider-graphs in the worst case. They also give a lower bound of $\Omega(n^{1/\log\log n}\log n)$ on the buffer size in so-called fork graphs (spider graphs where the hub node has degree 3). Note that in contrast to these results, we show a $\Theta(\log n)$ bound for any tree, albeit with a weaker adversary.

1.2 Our results

We start by pointing out that a slight variation of the *Local-Downhill* algorithm, shown in [1] to require buffers of size $\Omega(n)$ in paths, can in fact work with buffers of size $O(\sqrt{n})$, significantly improving upon the other local algorithms presented there.

A natural question to ask is whether this can be further improved. In this paper, we show a tight bound of $\Theta(\log n)$ on the buffer size needed by local algorithms for information gathering on directed paths and trees. On the one hand, we prove a lower bound of $\Omega(c\log n/\ell)$ (more precisely $c(1 + (\log n - 2\log\ell - 1)/2\ell)$) for the buffer sizes required by ℓ-local algorithms on directed paths of length n, where the injection rate of the adversary and the link capacity are both c. The lower bound also holds for bidirectional paths, albeit with a constant factor that is worse by a factor of 4. This improves on the result of [21] by working for a much weaker adversary, being significantly tighter, as well as applying to arbitrary constant locality ℓ.

On the other hand, for $c = 1$, we give local algorithms for directed paths and trees that require buffers of size $O(\log n)$. For the directed path, we give a very simple 1-local algorithm that achieves an upper bound of $\log n+3$, i.e. within a factor of 2 of the lower bound. In comparison with the algorithm from [21], our algorithm is simpler, achieves a better bound and works on a directed line, while the algorithm of [21] balances queues by sending packets away from the sink. For the directed tree, we give a very simple 2-local algorithm that achieves an upper bound of $O(\log n)$. This is in contrast with the lower bound of $\Omega(n)$ shown in [22] for spider graphs, emphasizing the difference between our adversary models.

To the best of our knowledge, the previous best local algorithm for convergecast in trees required buffer size $\Omega(n)$ in the worst case. While our algorithms are very simple to specify and implement, the analysis of both algorithms is based on a sophisticated book-keeping scheme which shows that for a buffer to build to size k, there must be an exponential number of nodes with smaller-sized buffers. All missing proofs are in the appendix.

2. NOTATION AND PRELIMINARIES

We consider tree networks of n nodes. The root of the tree, denoted s, is the sink node, which *consumes* packets. The nodes model hosts or routers in a communication network, and the edges represent communication links between them. Each edge can forward at most c packets along every outgoing link in every step. We consider an adversary of rate c; in every time step, the adversary injects a total of at most c packets at some nodes in the network. Our lower bounds work for any c, while our algorithms assume that $c = 1$. Since every packet is to be routed to the sink, the path taken by a packet is assumed to be the unique shortest path to the sink and does not need to be specified. As commonly assumed in the literature, we assume that time is divided into steps, each of which can be divided into 2 mini-steps. In the first mini-step, the adversary injects $\leq c$ packets into the network, and can choose the locations for the injections arbitrarily. In the second mini-step, each node uses its scheduling policy to forward at most one packet on each of its outgoing links.

For every node v, we denote by $s(v)$ its *successor* along the path to the sink. The *height* of a node v is the number of packets in its buffer, and is denoted by $h(v)$. A *configuration* C specifies the state of the network at the beginning of a given step. For our purposes, a configuration is specified by the heights of all nodes in the network. We denote the height of a node x in configuration C by $h_C(x)$. We assume that $h_C(s)$ is always 0. Let C be a configuration at the start of a step, and let C' be the configuration at the start of the following step. We will use shorthands $h(x)$ and $h'(x)$ for $h_C(x)$ and $h_{C'}(x)$, respectively. Throughout the paper, we

[1]Actually, the algorithm as it is formulated in [1] uses for $\rho > 1$ buffers of size $\Omega(\log\rho)$, as shown in Figure 2. However, it can be easily corrected by not activating a single path and taking ρ packets along it, but by having ρ activating steps. each applying to a single packet.

[2]They also study a ring network, in which this translates into clockwise and counter-clockwise direction towards the sink.

will denote by t the node into which the adversary injected a packet.

3. LOWER BOUNDS

In this section, we show lower bounds on the buffer size of ℓ-local algorithms for information gathering on paths; i.e. a node sees the buffer states of all other nodes up to hop distance ℓ, but not more.

THEOREM 1. *Any ℓ-local algorithm for information gathering on a directed path with link capacities c requires buffers of size $\Omega(c \log n/\ell)$.*

PROOF. Let n_0 be the largest number of form $l2^i$ that is smaller than n. The adversary works in stages. At the beginning of stage i, at time t_i, it assumes that there is a contiguous block B_i of nodes of size $K_i = n_0/2^i$ such that the average message density in B_i is at least $H_i = c(1+i/2\ell)$, i.e. the total number of messages M_i in the block B_i is at least $K_i H_i$. We show that as long as $K_i \geq 2\ell$, in $x_i = K_i/2\ell$ steps, the adversary is able to construct a block B_{i+1} of size K_{i+1} and average density H_{i+1}. This implies a lower bound of $\lceil H_{i'} \rceil$, where $i' = \log(n_0/2\ell)$ is the number of stages.

We start by showing that the assumption holds for stage $i = 0$. In each of the first n_0 steps, the adversary injects c messages at the leftmost node of the path. Set the initial block B_0 to be the leftmost n_0 nodes; i.e. $K_0 = n_0$ and $t_0 = n_0$. This yields $H_0 = c$, as none of the messages had time to travel outside block B_0.

Consider now the inductive step i.e. assume the inductive hypothesis holds for stage i. First, consider a scenario in which the adversary injects c messages at the rightmost node of B_i for $x_i = K_i/2\ell$ steps starting at time step $t_i + 1$. As the number of injected messages equals the available outflow from B_i, the number of messages in B_i cannot decrease.

Let M_r and M_l be the number of messages in the right and left half of B_i, respectively, at time $t_{i+1} = t_i + x_i$. By the inductive assumption it holds $M_l + M_r \geq K_i H_i$. If $M_r \geq H_{i+1}K_{i+1} = (H_i + c/2l)K_i/2 = H_i K_i/2 + cK_i/4\ell = H_i K_i + cx_i/2$, then the right half of K_i satisfies the condition for stage $i+1$ at time t_{i+1} and we are done. Otherwise, we have $M_l = H_i K_i - M_r \geq H_i K_i/2 - cx_i/2$.

Consider now an alternative scenario, in which the adversary instead injects messages into the leftmost node of B_i. As x_i is chosen in such a way that the information from the boundary of B_i is not able to reach the middle of B_i in time t_{i+1}, the flow of messages through the middle link is the same in both scenarios. Hence, the number of messages in the left half of B_i is now $M_l + cx_i \geq H_i K_i/2 - cx_i/2 + cx_i = H_{i+1}K_{i+1} + cx_i/2 = H_{i+1}K_{i+1}$.

Therefore, the adversary can always select a scenario in which the assumption for level $i+1$ are satisfied. This argument holds as long as $x_i \geq 1$, i.e. $K_i \geq 2l$. The number of stages is $\log(n_0/2l) = \lfloor \log(n/2l^2) \rfloor$, resulting in maximal buffer size of at least

$$c(1 + (\log n - 2\log l - 1)/2l) \in \Omega(c \log n/l)$$

□

COROLLARY 1. *If the insertion model allows for insertion of c messages with additional burstiness of δ [1], then then the adversary can force buffers of size $c(1 + (\log n - 2\log l - 1)/2l) + \delta \in \Omega(c \log n/l + \delta)$.*

PROOF. The adversary follows the same approach, and in the final stage adds an insertion burst of additional δ messages. □

A natural question is whether giving the algorithm the power to forward messages in both directions might help it to overcome the $\Omega(\log n)$ barrier. After all, the proof of Theorem 1 strongly relies on the fact that the messages were not able to "leak" out to the left. We answer this question in the negative and show below that using bidirectional links only reduces the constant factor in the lower bound:

THEOREM 2. *Any ℓ-local algorithm for information gathering on an undirected path with link capacities c requires buffers of size $\Omega(c \log n/\ell)$.*

4. 1-LOCAL ALGORITHM FOR PATHS

In this section, we give an optimal 1-local algorithm for buffer management that achieves information gathering on a directed path using $\Theta(\log n)$ buffer size, for injection rate and link capacity $c = 1$.

Recall that the local algorithms discussed in [1] have either unbounded buffer size (eg. *FIE*) or use buffers of size $\Omega(n)$ (*Downhill*, *Greedy*). In fact, a simple modification of *Downhill* achieves significant improvement to $O(\sqrt{n})$:

THEOREM 3. *Consider the local algorithm* Downhill-or-Flat *which forwards a packet whenever the buffer of its successor contains equal or smaller number of packets than its own buffer. Algorithm* Downhill-or-Flat *uses buffers of size $\Theta(\sqrt{n})$.*

Looking at the lower-bound examples given by Miller and Pat-Shamir for various local algorithms, we notice that:

- when the adversary injects at the left, the algorithm should efficiently (at throughput 1) forward messages to the right, otherwise the messages pile up on the left (*FIE* and *Downhill* fail in this). In particular, this suggests forwarding messages to the right if the buffer heights are equal.

- when the adversary injects at the right, the messages should not keep arriving from the left, otherwise they pile up on the right (*Greedy* fails in this, but also *Downhill-or-Flat*).

These two requirements seem contradictory, with no apparent way to satisfy them both. The main idea of our algorithm is to satisfy the first requirement for messages on odd heights, and the second one for even heights. At the moment the adversary starts injecting at the right, the packets start to pile up to the next height, switching to the "stopped" behaviour and spreading the piling up leftwards instead of up. If the adversary starts injecting on the left into stopped even-height nodes, the height raises to even and the packets start efficiently flowing to the right. In this way, the algorithm automatically adapts to the adversary's behaviour; this disarms the main tool the adversary uses to cause high buffer heights in the previously discussed local algorithms.

Before having a closer look at the behaviour of Algorithm Odd-Even, let us introduce some notation. Let us call a node an *up* node if its height went up, and a *down* node if its height went down, i.e. $h(x) < h'(x)$ for *up* node x and

Algorithm 1: Algorithm Odd-Even executed by node v

```
if If h(v) is odd then
    forward a packet to your successor s(v) iff
    h(s(v)) ≤ h(v)
else
    forward a packet to your successor s(v) iff
    h(s(v)) < h(v)
end
```

Algorithm 2: Creating a Balanced Matching

```
Set X to be the set of non-steady nodes of C', with the
  2up node (if any) treated as two consecutive up nodes.
while X contains at least two nodes do
    processing from the left, let x and y be the first two
    non-steady nodes in X.
    pair x with y and remove them from X
end
```

$h(x) > h'(x)$ when x is a *down* node; the nodes of unchanged height are *steady*. Note that as the link capacity is 1, the height of a *down* node is always reduced by 1, while an *up* node can have its height raised by 1 or by 2 (if it received from its predecessor and from the adversary, but did not send – at any round there can be at most one such node, called 2*up*). There is a special type of *up* node: the node that went up from 0 to 1, while all the nodes in front of it are of height 0. We will call it a *leading-zero* node. Note that there might not be an *leading-zero* node in the network.

Consider first a round in which the adversary did not inject any message. In such case, *up* and *down* nodes must alternate in the sense that the first node in any chain of sending nodes is always *down*, and the first node following this chain is always *up*[3]. The injection of a message by an adversary merely raises the height of the injected node by one, e.g. making an *up* node out of a *steady* one, or a 2*up* node out of an *up* one.

4.1 Balanced Matchings

In order to show that the heights of nodes do not go up too much, if the height of a node x goes up, we would like to "charge" that increase to another node y whose height went down in the same round. Intuitively speaking, this is as if y *gave* one if its packets to x.

We say that a non-steady node x is a *neighbour* of a non-steady node y iff there are only *steady* nodes between them.

Definition 1. *A set P of node pairs is a* balanced matching *for a configuration C' iff*

- *every* up *node is paired with a neighbouring* down *node, except possibly for the* leading-zero *node*
- *every* down *node is paired with a neighbouring* up *or* 2up *node, except possibly the rightmost* down *node*
- *the* 2up *node, if any, is paired with its two neighbouring* down *nodes*
- *no* steady *node is paired with another node*

These possible pairs (and one triple) will be called *down-up*, *up-down* and *down-2up-down* intervals, based on the type of nodes when traversing from the left. In what follows, the *down-2up-down* interval will implicitly be treated as a *down-up* interval followed by an *up-down* interval.

Claim 1. *At most one non-steady node remains unmatched after executing Algorithm 2, and it is either the rightmost* down *node, or the* leading-zero.

[3]Note that this holds for any algorithm on the directed line sending at most one message at a time

Proof. First, note that Algorithm 2 fails to make *up-down* or *down-up* pairs only if there are three consecutive *down* or *up* nodes. However, this never happens: If there is no injection, the *down* and *up* nodes alternate. If there is an injection at node t, it can either make a *steady* node out of a *down* node, make an *up* node out of a *steady* node or make a 2*up* node out of an *up* node. In any case, as before the injection the *up* and *down* nodes alternated, at most two consecutive *up* nodes are created and there are no two consecutive *down* nodes – see Figure 3.

As each iteration of the while loop removes two non-steady nodes, only the rightmost non-steady node remains unmatched, and only in case the number of non-steady nodes was even (counting the 2*up* node as 2 and the *down*-and-injected node as 0). Hence, it remains to be shown that if the remaining node is an *up* node, it must be a *leading-zero*.

If there is *leading-zero*, by its definition it is the rightmost *up* node and we are done.

Consider the chain of sending nodes ending in the sink. If there is no *leading-zero* node, the neighbour of the sink must be of non-zero height and hence this chain is non-empty. If there is no injection into this chain, the first node of this chain must go down and being the rightmost non-steady node, the lemma holds.

Finally, if there is injection into this chain, as the *down* and *up* nodes alternate for non-injection case, before the injection the number of non-steady nodes was odd (starting with *down* and finishing with *down*). The injection either creates a rightmost *up* node (if inserted inside the chain), which will pair with the *down* node at the beginning of the chain, or it transforms the rightmost *down* node into a *steady* one. In either case, no unpaired non-steady node remains. □

Lemma 1. *Algorithm 2 creates a balanced matching.*

Proof. Consider the processing of X in the while loop. As the *up* and *down* nodes alternate, starting with a *down* node, *down-up* intervals are created before encountering the injected node. If an injection creates two neighboring *up* nodes, switching to *up-down* intervals starting at the injected node takes care of all the remaining non-steady nodes, with Claim 1 taking care of the last non-steady node, if there is any. Note that by construction no steady node is paired.

The possible cases, together with their handling are shown in Figure 3. □

The pairs of the balanced matching will be called *matching pairs*.

The adversary could conceivably create a high-height node v by first cheaply creating a lot of low-height nodes and then charging those while increasing the height of v; we prevent that be requiring $h_C(y) \geq h_C(x)$. The next lemma shows

that this requirement, as well as monotonicity of the intervals between the nodes of the matching pairs, is indeed satisfied:

LEMMA 2. *Let (x_d, x_u) be a matching pair with x_u being the* up *node of this pair. Then $h(x_u) \leq h(x_d)$.*

Moreover, if (x_d, x_u) is a down-up *interval, then $h(z) \geq h(s(z))$ for all nodes $z \neq x_u$ between x_d and x_u, and if (x_u, x_d) is an* up-down *interval, then $h(z) \leq h(s(z))$ for all nodes $z \neq x_d$ between x_u and x_d.*

PROOF. Let us first consider the case of (x_d, x_u) being a *down-up* interval, i.e. x_d is behind x_u. As x_d went down, $x_d \neq t$ and it sent a message to $s(x_d)$), i.e. $h(x_d) \geq h(s(x_d))$. As none of the nodes between x_d and x_u changed their height, each one of them must have received and sent a message[4]. Combining with the fact that in any chain of sending nodes, the node heights are non-increasing yields the lemma statement.

If (x_u, x_d) is an *up-down* interval, then x_d has sent to $s(x_d)$, but received nothing from its predecessor $pr(x_d)$. If none of the nodes from x_u to $pr(x_d)$ has sent a message, then their heights form a non-decreasing sequence and the lemma holds. However, there cannot be a node x' between x_u and x_d that has sent a message – the first non-steady successor of such a node would be an *up* node, violating the definition of *up-down* interval. □

4.2 Attachment Scheme

If $h(x_d) > h(x_u)$, the adversary pays for raising the height of x_u by lowering the height of a costlier, higher height node, a net loss. However, the case of $h(x_d) = h(x_u)$ allows the adversary to raise a node height without losing the effort invested into another node of higher height. The core of the proof is to show that in Algorithm Odd-Even such a situation cannot occur too often. To accomplish this, when x_u charges to x_d and $h(x_d) = h(x_u)$, we take note that x_d "gave" x_u a packet by *attaching* x_d with the new $h'(x_u)$-th packet of x_u. This attachment will remain until either x_d changes height or until x_u loses its $h'(x_u)$-th packet.

In this way, creating a node x of height $h+1$ uses up two nodes of height h: both x itself (as it is no more of height h), but also a node y that attached to x. Taken to its conclusion, this means that creating a node of height h incurs cost exponential in h – provided the attached node is not reused for another *up* node. The key observation (and design goal) is that once a node y gives away a packet and becomes attached, it cannot be charged again by another *up* node as long as this attachment persists. Such a y then becomes useless to the adversary, and we will thus call y a *residue* - it is a leftover that resulted from the creation of a node of higher height.

In order to maintain proper bookkeeping, some attachments may need to be "passed" to other nodes. To keep track of the fact that y virtually gives a packet to x, we create a pointer from x to y. However, when y virtually received this packet from some other node z in a previous step, it would have had a pointer to z; we retain this history by directly creating a pointer from x to z. Intuitively, the greater the height of a node, the more pointers it will have. We formalize all this in the notion of an *attachment scheme* defined below.

[4]Observe that if a node sends a message and receives injection, it is not included in the balanced matching

For $i \geq 3$, a packet $x[i]$ has $i-2$ available *slots* denoted $x[i,1], x[i,2], \ldots, x[i,i-2]$. Note that every slot $x[i,j]$ satisfies $1 \leq j \leq h(x) - 2$.

DEFINITION 2. *An* attachment scheme *A for a configuration C is a set of ordered pairs of the form $(x[i,j], y)$, where $x[i,j]$ is a packet slot and y is a node distinct from x, such that*

1. *$j = h(y)$;*
2. *each packet slot or node is attached to exactly one element, i.e. for any $(x'[i',j'], y') \in A$ distinct from $(x[i,j], y)$, we have $x[i,j] \neq x'[i',j']$ and $y \neq y'$;*

If $(x[i,j], y) \in A$, we say that slot $x[i,j]$ is attached to node y, and similarly that node y is attached to slot $x[i,j]$ (we may also say $x[i,j]$ is the *guardian* of a *residue* y, as will be explained later). We may sometimes write (x,y) instead of $(x[i,j], y)$ when the values of i,j are irrelevant. The node y attached to $x[i,j]$ is denoted $att_A(x[i,j])$.

Figure 4 illustrates a node x with all its available packet slots attached to a node of appropriate height.

LEMMA 3. *Let A be an attachment scheme for a configuration C. Let $m := \max_{x \in V} h_C(x)$, and assume $m \geq 3$. Then there are at least $2^{m-2} - 1$ distinct nodes that are a residue of A.*

PROOF. For a node $x \in V$, denote by $R'(x)$ the set of residues of A that are attached to a packet slot of x, i.e. $R'(x) = \bigcup_{3 \leq i \leq h(x)} \bigcup_{1 \leq j \leq i-2} \{att_A(x[i,j])\}$. Define $R(x)$ inductively as follows: if $h(x) \leq 3$, then $R(x) = R'(x)$, and if $h(x) \geq 4$, then $R(x) = R'(x) \cup \bigcup_{x' \in R'(x)} R(x')$. That is, $R(x)$ is the set of residue nodes that are attached to x directly or indirectly. Observe that for any two x and x' with $h(x) = h(x')$, we have $|R(x)| = |R(x')|$.

For an integer p, denote by $r(p)$ the cardinality of $R(x)$ for a node x such that $h(x) = p$. We show that $r(p) = 2^{p-2} - 1$. We have $r(1) = r(2) = 0$, since nodes of height 1 or 2 have no available slots for residues. For $p \geq 3$, a node x with $h(x) = p$ has a packet $x[p]$ with each slot $x[p,1], \ldots, x[p,p-2]$ attached to a residue, and there are $p-2$ of those. Also, for each $1 \leq i \leq p-2$, the residue $att_A(x[p,i])$ implies the existence of $r(i)$ other residue nodes. Moreover, the packet $x[p-1]$ implies the existence of $r(p-1)$ residue nodes. Note that by Rule 2 of attachment schemes, no residue is double-counted. Thus, we get

$$\begin{aligned} r(p) &= p - 2 + \sum_{i=1}^{p-1} r(i) \\ &= p - 2 + r(p-1) + \sum_{i=1}^{p-2} r(i) \\ &= p - 2 + r(p-1) + r(p-1) - (p-3) \\ &= 1 + 2r(p-1) = 2^{p-2} - 1 \end{aligned}$$

when $r(2) = 0$ (the third equality is due to $r(p-1) = p-3+\sum_{i=1}^{p-2} r(i)$). The Lemma follows by setting $p = m$. □

LEMMA 4. *Let A be an attachment scheme for a configuration C. Then $\max_{x \in V} h_C(x) \leq \log n + 3$.*

PROOF. By Lemma 3, if $m := \max_{x \in V} h_C(x)$, then there are at least $2^{m-2} - 1$ nodes that are a residue. Since all these nodes are distinct, we have $2^{m-2} - 1 \leq n$, which yields $m \leq \log n + 3$. □

4.3 Maintaining an Attachment Scheme

If for every configuration C, there exists an attachment scheme, it follows from Lemma 4 that the height of every node in the path is always upper bounded by $\log n + 3$. We now proceed to show by induction that every configuration indeed admits an attachment scheme. The initial configuration consists of height 0 (i.e. slot-free) nodes, hence it vacuously admits an attachment scheme. If a configuration C admits an attachment scheme, then we will show that the next configuration C' also admits one. The transition from C to C' is done by handling separately and independently the matching pairs of C': We present an algorithm which processes a matching pair $\{x_u, x_d\}$ by changing the heights of its nodes to their new values in C' and rearranging some attachments coincident with x_u or x_d so that an attachment scheme is maintained.

In order to carry out the inductive step, we need to strengthen the definition of an attachment scheme:

DEFINITION 3. *An attachment scheme A is* valid*, if in addition to Rules 1 and 2, the following rules are satisfied for each residue y and its guardian x of A:*

3. *if $h(y)$ is even, then x is in front of y;*
4. *if $h(y)$ is odd, then x is behind y;*
5. *for every node z on the path between x and y, $h(z) \geq h(y)$.*

Let P be a subset of matching pairs of a balanced matching of C'. We say that C_P is an *intermediate configuration* for P iff $\forall x, x \in P : h_{C_P}(x) = h(x)$, while $\forall x, x \notin P : h_{C_P}(X) = h'(x)$.

We will need the following three technical lemmas:

LEMMA 5. *If (x_u, x_d) is a matching pair with $h(x_u) = h(x_d) = h$ then x_u is not a residue.*

PROOF. First note that by Lemma 2, $h(z) = h$ for every node between x_u and x_d. Now, assume, by contradiction, that x_u has a guardian z.

If x_d is behind x_u, h must be odd, and x_u is attached to a slot $z[i, l]$ with z behind x_d. Since x_d goes down, it does not receive from its predecessor p and thus $h(p) < h(x_d)$. But p is on the $z - x_u$ path and $h(p) < h(x_u)$, and so A contradicts Rule 5.

If x_d is in front of x_u, then h must be even and x_u must be attached to a slot $z[i, l]$ with z in front of x_d. As x_d goes down, it sends to $s(x_d)$ and so $h(s(x_d)) < h(x_d)$. As $s(x_d)$ is on the $z - x_u$ path, we again conclude that A contradicts Rule 5. □

The following lemma is crucial in proving that the residues are not shared:

LEMMA 6. *Let y be a residue of A. Then y is not a* down *node.*

PROOF. If y receives a packet from the adversary, then y cannot go down, so assume this is not the case. If $h(y)$ is even, then by Rule 3 and Rule 5 of the rules of an attachment scheme, the successor $s(y)$ of y must satisfy $h(s(y)) \geq h(y)$. Thus y does not send forward since it is even, and so it cannot go down. If $h(y)$ is odd, then by Rule 4 and Rule 5, y is preceded by a node z such that $h(z) \geq h(y)$. Then z sends to y (even if $h(z) = h(y)$ since this height is odd), and so y cannot go down. □

Algorithm 3: Processing a balanced matching.

Input : Configurations C and C' and an attachment scheme A for C
Output: An attachment scheme A' for C'', where C'' differs from C' (and equals C) only for the possible *down-2up-down* triple, the *leading-zero* and the unmatched rightmost *down* node

Let M be a balanced matching for C'
Set $P := M$ and $A' := A$
while $P \neq \emptyset$ **do**
  Let (x_d, x_u) be a matching pair from P
  Set $A' := processPair(C_P, A', x_d, x_u)$;
  Set $P := P \setminus \{x_d, x_u\}$;
end
Return (A')

Algorithm 4: Handling a matching pair.

function *processPair* (C_P, A_P, x_d, x_u);
Input : An intermediate configuration C_P, an attachment scheme A_P for C_P, and a matching pair $(x_d, x_u) \in P$ with x_d and x_u being the *up* and *down* nodes, respectively.
Output: An attachment scheme $A_{P'}$ for $C_{P'}$, where P' is obtained from P by removing (x_d, x_u).
Let $h_d := h(x_d)$ and $h_u := h(x_u)$
Let $A' := A_P$.
if *there is a slot* $x_d[i, h_u]$ *such that* $(x[i, h_u], x_u) \in A'$ *and* $i \neq h_d$ **then**
  Swap the $x_d[i, h_u]$ and $x_d[h_d, h_u]$ attachments: in A', replace $(x_d[i, h_u], x_u)$ by $(x_d[i, h_u], att_{A_P}(x_d[h_d, h_u]))$ and replace $(x_d[h_d, h_u], att_{A_P}(x_d[h_d, h_u]))$ by $(x_d[h_d, h_u], x_u)$. ;
  `// Here we ensure that when` x_u `gets detached, it does not leave slot` $x_d[i, h_u]$ `empty`
end
Pass all possible attachments from the $x_d[h_d]$ packet to the $x_u[h_u + 1]$ packet and remove the others, i.e. remove from A' all the attachments $\{(x_d[h_d, i], att_{A_P}(x_d[h_d, i])) : 1 \leq i \leq h_d - 2\}$ and add to A': $\{(x_u[h_u + 1, j], att_{A_P}(x_d[h_d, j])) : 1 \leq j \leq \min(h_d - 2, h_u - 1)\}$
if $h_d = h_u$ *and* $h_d \geq 2$ **then**
  Add $(x_u[h_u + 1, h_u - 1], x_d)$ to A'
end
if x_u *is a residue of* A_P **then**
  Let $z[i, h_u]$ be the packet slot attached to x_u in A'
  Remove the $(z[i, h_u], x_u)$ attachment from A'
  if $h_d = h_u + 1$ **then**
    Add to A' the attachment $(z[i, h_u], x_d)$
  else if $h_d \geq h_u + 2$ *and* $z \neq x_d$ **then**
    Let $y = att_{A'}(x_d[h_d, h_u])$
    Add to A' the attachment $(z[i, h_u], y)$
end
Return A' as $A_{P'}$

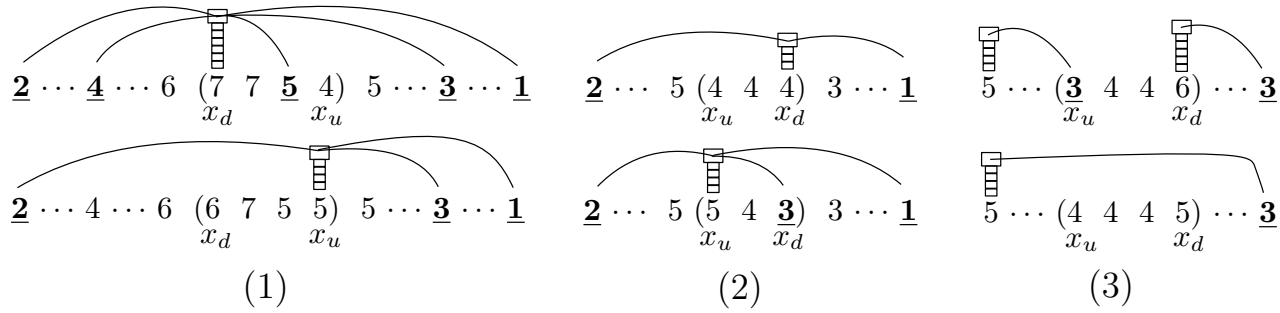

Figure 1: Three examples of applying Algorithm 4. Top: the state before, bottom: the state after. The parentheses surround the processed matching pair. We only represent the packets, attachments and residues of interest. (1) A down-up interval illustrating how x_d passes all possible attachments to x_u (line 7 of Algorithm 4). Note that the residues of value 4 and 5 gets detached in $C_{P'}$. (2) An up-down interval in which $h_d = h_u = 4$. Here x_d passes all its attachments, in addition to becoming a residue attached to x_u (line 9). (3) An up-down interval in which x_u was a residue attached to some slot $z[i, h_u]$ (here z is the node of value 5), and $x_d[h_d, h_u]$ is attached to a node y (y is the node of value 3). After processing, y is attached to $z[i, h_u]$ (line 18).

LEMMA 7. *The following facts hold when Algorithm 4 processes a matching pair (x_d, x_u):*

1. *after being processed, no* up *node remains a residue of another node*
2. *no existing slot has become empty*
3. *no new empty slot has been created*
4. *whenever an attachment to residue y is transferred*
   a) *from x_d to x_u on line 7*
   b) *from z to x_d on line 15*
   c) *from x_d to z on line 18*

   $h'(w) \geq h'(y)$ holds for all nodes between the nodes transferring the attachment (endpoints included)
5. *the relative order (in front of, or behind) between residues and their guardians never changes*

This allows us to prove that all the rules of the attachment scheme are satisfied:

LEMMA 8. *Processing one pair by Algorithm 4 maintains all the rules of the attachment scheme.*

PROOF. 1. A new attachment created on line 9 satisfies Rule 1, as it is attached to the slot $h_u - 1 = h'_d$, since x_d went down from height $h_d = h_u$. Furthermore, whenever an attachment is swapped or transferred, the slot position never changes, as can be seen by examining the lines 5, 7, 15 and 18. As new attachments are created only on line 9, the Rule 1 follows for residues that do not change their height. As (by Lemma 6) the residues don't go down, and the attachment of a residue that went up is removed on line 13. Rule 1 is maintained in every case.

2. The fact that each residue has a single guardian follows from the combination of Lemma 6 with the fact that a new residue is created only on line 9 from a *down* node. The fact that each slot attaches to a single residue follows from the fact that the slots gaining a residue on line 7 are new and had no attachments, and the fact that the slot getting a residue on line 9 is also a new one, but did not get a residue on line 9. Also, no empty slots are created, as shown in Lemma 7.

3. Note that when an odd-height residue is created, it must have been $h_u = h_d$ (as only then are the residues created) with h_d even (as x_d became residue after going down and becoming odd). As x_u was even and went up, this must have been an *up-down* interval, i.e. x_d (the residue) is in front of x_u (its guardian). The Fact 5 then ensures that this order is maintained.

4. Analogously as the previous rule, but h_d was odd, hence this must have been *down-up* interval and therefore x_d is behind x_u.

5. Follows by combining Lemma 2 (when the attachment is created), with Rule 5 and Fact 4 inductively maintaining this Rule. Note that breaking Rule 5 for a guardian-residue pair (x, y) not involved in processing is also impossible: A node x_d between x and y can go down only if (1) $s(x_d)$ is already smaller (violating the inductive hypothesis about Rule 5) or (2) x_d is of odd height and its predecessor $pr(x_d)$ is already smaller – as (x, y) are not involved, $x_d \notin \{x, y\}$, again violating Rule 5.

□

We are now ready to prove the upper bound on buffer sizes for paths.

THEOREM 4. *Algorithm Odd-Even uses buffers of size at most* $\log n + 3$.

PROOF. It follows from Lemma 8 that processing all pairs of a balanced matching by Algorithm 4 (including the two pairs concerning *down*-2*up*-*down* interval) maintains a valid attachment scheme. What remains to be dealt with is the right-most *down* node and the *leading-zero* node. The last one is not a problem, as it was of height 0 and hence not a residue, nor does it have a packet slot, as it is of height 1.

The right-most *down* node could have only released some attachments, and did not gain any, so it does not need any sophisticated (creation/passing) attachment processing (by Lemma 6 it was not a residue, so no empty slots were created either).

Note that handling the *down*-2*up*-*down* interval as a sequence of two intervals sharing an *up* node is perfectly fine: from the point of the right pair this looks the same as if t was of height $h(t)+1$ and received a message from the left. Lemma 4 now completes the proof of the theorem. □

5. 2-LOCAL ALGORITHM FOR TREES

The first observation is that lookahead of 1 is not sufficient: Consider the graph from Figure 2, with node u having $\sqrt{n}$ neighbours and the same schedule as discussed in its caption. When the packets arrive simultaneously to v's, each v_i will send a packet to u, forcing u to need buffer of size $\sqrt{n}$.

Hence, we consider a 2-local algorithm. The algorithm is a straightforward generalization of Algorithm Odd-Even:

Algorithm 5: Algorithm Tree

```
1 if the height h of the node is odd then
2 |   forward a packet to your successor iff its height is at
  |   most h and you have the highest priority among
  |   your siblings
3 else
4 |   forward a packet to your successor iff its height is
  |   less than h and you have the highest priority among
  |   your siblings// even height h
5 end
```

The algorithm is completed by specifying the priority scheme: A sibling with a higher height has higher priority. Among the siblings of the same maximal height, choose arbitrarily.

Let us now introduce more nomenclature. An internal node v of in-degree at least 2 will be called an *intersection*. For a fixed round, in each intersection there will be at most one incoming packet; the branch where it comes from will be called a *priority line*[5]. A non-priority line ends in a *blocked* node. Hence, the tree can be viewed as a set of lines, starting in leaves and ending in blocked nodes, with one branch, called *drain* making it all the way to the sink. One of the lines might contain the injected node – we will call it the *injected* line. All other lines are *normal*. Note that the *up* and *down* nodes on non-injected lines alternate, starting with a *down* node (exactly like in paths) and ending with a *leading-zero* or *down* node if the line is a *drain*, otherwise ending with an *up* node.

The first step is a generalization of balanced matchings to trees: The matchings for each normal line translate directly (they are each just a collection of *down*-*up* intervals from left to right). If the injected line is also the *drain* one, this is handled as in the single line case with injection. But if the injected and *drain* lines are different, let v be the intersection on which the injected line blocks. As each normal blocked line has an equal number of *up* and *down* nodes, the injected line has an excess of one *up* node: Applying Algorithm 2 leaves it with the rightmost *up* node x unpaired. At this moment, it is impossible to carry on constructing balanced matching as a union of balanced matchings of the lines: We need to introduce *crossover pairs* containing nodes from different lines. This is what is done in the while loop: As the last non-steady node y of the priority line is *down*, we pair x with y to form a *crossover pair*. Since we have removed y from its line, we need to re-do its pairings that were in front of y, switching to *up*-*down* intervals. This possibly leaves another unmatched *up* node at the end, which needs to be handled in the same manner. We make these crossover pairs until we eventually reach the drain, where the *up*-*down* matchings do not leave an unmatched *up* node at the end. An example of applying Algorithm 6 is shown in Figure 5.

Algorithm 6: Balanced Matching on a Tree

```
1  For each line, apply the balanced matching algorithm
   for paths:
2  if the injection was on the priority line to the sink then
3  |   we are done, nothing left to do
4  else
5  |   while there is an unmatched up node x_u do
6  |   |   Let v be first intersection in front of x_u, and let
   |   |   p_v be the priority line containing v.
7  |   |   Let x_d be the first down node behind v on p_v.
8  |   |   Remove the pairs (including the one containing
   |   |   x_d) in front of x_d on the line of x_d
9  |   |   Add (x_d, x_u) to the set of matching pairs
10 |   |   Process the remainder of the x_d's line using the
   |   |   algorithm for paths (i.e. add up-down pairs
   |   |   while possible)
11 |   end
12 end
```

Hence, a tree-version of Lemma 1 holds:

LEMMA 9. *Algorithm 6 creates a balanced matching.*

We will often make use of the following simple property of matching pairs.

LEMMA 10. *Let x_u be an* up *node lying on a priority path p that is not the drain, and let v be intersection node on which p does not have priority. Then x_u is matched with a node x_d behind v.*

PROOF. If (x_d, x_u) is not a crossover pair, they must belong to the same priority line p, implying that x_d is behind v. If (x_d, x_u) is a crossover pair, by construction x_d is a *down* node behind v. □

In paths, the notion of *between* two nodes is straightforward. In trees, we will generalize it to fit our purpose: *between* x and y is satisfied by all nodes on the path from x to y, *except* for the node v (if any) in which this path changes direction from forward to backward. This node will be called the *tip* of the crossover pair.

Before introducing the tree-version of Lemma 2 we need a bit more notation: Let (x, y) be a crossover pair with tip v. $p_v(z)$ will denote the predecessor of v on the path from z to v. If clear from the context, we will omit the subscript v.

We now show a tree-version of Lemma 2:

[5]It can happen that the intersection has no incoming packet. In such a case, we choose as the priority line, the line into which there was an injection; if no such line is behind, select arbitrarily.

LEMMA 11. *Let (x_d, x_u) be a matching pair with x_u being the* up *node of this pair. Then $h(x_u) \leq h(x_d)$ and $h(z) \geq h(x_u)$ for all nodes z between x_u and x_d.*

Moreover, the nodes on the path from x_d to x_u appear in non-increasing order of height, with the possible exception of the tip v between x_d and x_u.

PROOF. The lemma statement holds directly for non-crossover matching pairs, as it translates directly from the path case. Hence, let (x_u, x_d) be a crossover pair. By construction, x_u is in the blocked injected line, while x_d is in the priority path, w.r.t. to intersection v. As the priority path had priority for sending to v, $h(p(x_d)) \geq h(p(x_u))$. Using the same arguments as in Lemma 2, we have that every node on the $x_d - p(x_d)$ path sends forward, implying monotonically decreasing heights on this path. Also, none of the nodes on the $x_u - p(x_u)$ path sends (otherwise, x_u would not be the last node on its priority path), and so by traversing backwards from $p(x_u)$ to x_u, the nodes heights appear again in a monotonically decreasing order. This proves the lemma also for the new meaning of *between* for crossover nodes. □

The attachment scheme is defined analogously as for the path case. However, in order to limit technicalities, we limit Rule 2 to residues of even value. This implies that Lemmas 3 and 4 yield a $2\log n + O(1)$ bound.

The Rules 3, 4 and 5 are replaced as follows:

DEFINITION 4. *For each pair (x, y) of an attachment scheme, where y is a residue and x is its guardian, the following rules must be satisfied:*

6. *if $h(y)$ is even, x is not behind y;*
7. *if (x, y) is not a crossover pair, then $h(z) \geq h(y)$ holds for every node z on the path between y and $p(y)$; otherwise if (x, y) is a crossover pair, $h(z) \geq h(y)$ holds for every node z on the path between y and $p(y)$, and $h(z) > h(y)$ holds for every node z on the path between x and $p(x)$.*

This allows us to prove (using the same arguments; note that the proof is not valid for odd-height residues) the tree-version of Claim 6:

CLAIM 2. *Let x be an even-height residue of A. Then x does not go down.*

In the rest of the proof, when we discuss residues and attachments, we limit ourselves to even height residues and corresponding attachment pairs.

First, we show that Lemma 5 holds also for trees. As this was the only necessary ingredient for Fact 2, this implies that after running Algorithm 4 on every matching pair, the resulting attachment scheme is still full.

LEMMA 12. *If (x_u, x_d) is a matching pair with $h_u = h_d = h$, then x_u is not a residue.*

PROOF. Suppose that x_u is a residue, with z as its guardian. Observe that h must be even, and that the outneighbor $s(x_d)$ of x_d must satisfy $h(x_d) > h(s(x_d))$. Since $h(x_d) = h(x_u)$, by Lemma 11 $s(x_d)$ cannot be between x_d and x_u, which is only possible if (1) x_u is behind x_d, or (2) x_d and x_u are on different priority paths, and $s(x_d)$ is the intersection of these paths.

Suppose first that (1) holds. Then by Rule 7, $s(x_d)$ cannot be behind z, so (x_u, z) must form a crossover attachment with tip v. Moreover, $s(x_d)$ cannot be behind v. If $s(x_d) = v$, then $h(p_v(z)) > h(p_v(x_u)) = h(x_d)$ (by the second part of Rule 7). But then, x_d does not have priority on $s(x_d)$, and so it could not go down. If $v = x_d$, then x_d receives from $p_v(z)$ (or another node) and cannot go down. Finally if v is behind x_d, then $h(p_v(z)) > h(p_v(x_u))$ implies that x_u and x_d are not on the same priority line, and so the (x_u, x_d) matching is impossible, since by construction all non-crossover pairs are on the same priority line. Thus (1) cannot occur.

Assume now that (2) holds. Then by the second part of Rule 7, z cannot be behind x_d. Also, $s(x_d)$ cannot be between x_u and z, and z cannot be between x_u and x_d (the latter by Lemma 11). These imply that (x_u, z) must form a crossover attachment with the tip v lying on the path between x_u and $s(x_d)$, with $v = s(x_d)$ being possible. But in the latter case, as before x_d does not have priority on $s(x_d)$ and couldn't go down, so assume $v \neq s(x_d)$. Since $h(p_v(z)) > h(p_v(x_u))$, x_u is blocked by v, and by Lemma 10, x_u would be matched with a node behind v, which is not the case for x_d. □

The proofs of Facts 1, 2 and 3 of Lemma 7, as well as the proofs from Lemma 8 that Rules 1 and 2 are satisfied are based on the behaviour of Algorithm 4, using in addition only Claim 6 and Lemma 5; using Claim 2 and Lemma 12 instead, the same proofs apply to trees without need for any modifications.

We prove that, after running Algorithm 4 on a single matching pair, crossover or not, Rules 6 and 7 are satisfied directly (here we do not refer to Facts 4 and 5). As before, $h(x)$ is the height of a node at the start of the round, and $h'(x)$ its height after the round.

We first show that unmodified attachments are still valid, then proceed with the new attachments created by the algorithm.

LEMMA 13. *Let (x, y) be an attachment of A that has not changed after running Algorithm 4 on a matching pair. Then (x, y) still satisfies Rules 6 and 7.*

PROOF. As (x, y) did not change, Rule 6 is satisfied by induction hypothesis for A. As for Rule 7, if (x, y) is not crossover, no node w between x and y with $h(w) = h(y)$ can go down, since such a w would be even and have its outneighbor $s(w)$ with $h(s(w)) < h(w)$, contradicting Rule 7.

However, if (x, y) is crossover with tip v, Rule 7 could conceivably be violated by a node w going down between x and $p(x)$, or between y and $p(y)$. In the latter case, since w is followed by a node $s(w)$ of strictly smaller height, only $w = p(y)$ is possible. Let $h = h(y)$. As $h(p(x)) \geq h + 1$, for $p(y)$ to go down, it must have been of height at least $h + 1$ to get priority over $p(x)$, i.e. it won't go down below h. Hence, the only way Rule 7 could be violated is by w between x and $p(x)$ going down from $h + 1$ to h. As h is even, $h + 1$ is odd. By Rule 7 all the nodes w between x and $p(x)$ are of height at least $h + 1$, with x being of height $h + 2$ by construction of the attachment pair. As the pair (x, y) did not change, x did not go down, hence w cannot go down and Rule 7 is maintained. □

LEMMA 14. *Let (x_u, x_d) be a new attachment created on line 9. Then (x_u, x_d) satisfies Rules 6 and 7.*

PROOF. Recall that in this situation, $h(x_d) = h(x_u)$. If (x_u, x_d) is a non-crossover attachment, the rules are satisfied in the same way as in the path. If it is a crossover attachment, Rule 6 is vacuous. For Rule 7, Lemma 11 implies that every node w between x_d and x_u has $h(w) = h(x_u) > h'(x_d)$. Rule 7 follows, since for $w \neq x_d$, $h'(w) \geq h(w)$. □

LEMMA 15. *Let (x_u, y) be an attachment formed by passing y from x_d to x_u on line 7 of Algorithm 4. Then (x_u, y) satisfies Rules 6 and 7.*

PROOF. For Rule 6, suppose that y is in front of x_u. Because y cannot be in front of x_d, y must be between x_u and x_d (note that y cannot be the intersection node between x_u and x_d, as this would put y in front of x_d). But since $h(x_d) \geq h(x_u) \geq h(y) + 1$, this contradicts Lemma 11.

For Rule 7, we first show that every node w between x_u and y satisfies $h(w) \geq h(y)$. If w is between x_d and x_u, we have $h(w) \geq h(x_u) > h(y)$ and if w is between x_d and y we have $h(w) \geq h(y)$. The only way w is not on any of these paths is if w is a common intersection between x_u, x_d and y. But in this case, by definition w is not between x_u and y.

Thus if Rule 7 fails, (x_u, y) is a crossover attachment. All possible positions of x_u w.r.t. to x_d and y that may cause problems with the last part of Rule 7 are shown in Figure 6. First, observe that $h(x_u) \geq h(y) + 1$ and for all nodes w between x_d and x_u, we have $h(w) \geq h(y) + 1$. Furthermore, the same is true for nodes between x_d and v, where v is the tip of (x_d, y), in the case of a crossover attachment. These two clauses cover all possible segments from x_u to the tip of (x_u, y), as can be straightforwardly verified in Figure 6 – any backwards segment of the path from y to x_u is either a part of the original backwards segment of the path from y to x_d or a (backwards) segment of the path from x_d to x_u.

This concludes the proof. □

LEMMA 16. *Let (z, x_d) be an attachment formed by swapping the residue of z from x_u to x_d on line 15 of Algorithm 4. Then (z, x_d) satisfies Rules 6 and 7.*

PROOF. Suppose first that Rule 6 fails. All possible cases of z not being behind x_u while being behind x_d are shown in Figure 7: The case a) is impossible, because the heights of the nodes between the matching pair (x_u, x_d) form a monotone sequence, while $h(z)$ sticks out: $(h(x_d) = h(x_u) + 1$, but $h(z) \geq l + 2)$.

The case b): By the last part of Rule 7, all nodes between z and $p(z)$ are of height at least $h(x_u) + 1$. As x_d is of odd height $h(x_u) + 1$, it can't go down.

The case c) is also prohibited: since $h(p(z)) > h(p(x))$, x_u and x_d are not on the same priority line and would not be matched together.

For Rule 7, similarly as in the previous Lemma, we show that every node w between z and x_d must have $h(w) \geq h(x_d)$. This follows from the fact that every node w between x_u and z, or between x_u and x_d, must also satisfy $h(w) \geq h(y)$. So assume that (z, x_d) is a crossover attachment with tip v, and that there is a node w on the $p(z) - z$ path having $h(w) = h(x_d)$. In this case, x_u cannot be behind z, because if so, the $x_u - x_d$ path contains the nodes x_u, z, w, x_d in this order, which contradicts the monotonicity prescribed by Lemma 11. Moreover by Rule 7, x_u cannot be in front of z, and (z, x_u) could not have been a crossover attachment with a tip v' in front of w. Thus (z, x_u) is a crossover attachment with tip v' behind w or $v' = w$. In both cases, $p(x_u)$ does not have priority on v', and by Lemma 10, x_u should be matched with a *down* node behind v', which is not the case for x_d. □

LEMMA 17. *Let (z, y) be an attachment formed on line 18 of Algorithm 4. Then (z, y) satisfies Rules 6 and 7.*

PROOF. Recall that in this situation, the previous attachments (x_d, y) and (z, x_u) are removed and replaced by (z, y) (hence $h(x_u) = h(y)$). For Rule 6, suppose that z is behind y. Observe that since x_d was attached to y, x_d is either in front of y, or the tip v' of (x_d, y) is in front of y. Thus if x_u is behind z, then the $x_u - x_d$ path first contains z, then y, contradicting monotonicity. Thus assume x_u is not behind z. Then (z, x_u) is a crossover attachment, say with tip v. By Rule 7, v cannot be in front of y, and thus v is either behind y or $v = y$. But then, $p_v(x_u)$ does not have priority on v, and since x_d (or v') is in front of y, by Lemma 10 x_u would not be matched with x_d.

Now for Rule 7. Suppose there is a node w between y and z with $h(w) < h(y)$. Then x_u cannot be behind w (Rule 7 on (z, x_u)) and x_d must be behind w (Rule 7 on (x_d, y)). This implies that w is between x_d and x_u, contradicting Lemma 11 since $h(w) < h(y) = h(x_u)$. Thus if Rule 7 fails, (z, y) is a crossover attachment with tip v' and there is a node w on the $p(z) - z$ path with $h(w) = h(y)$. In this case, we must have x_u behind w (x_u is not in front of w by Rule 6, and cannot be in a crossover attachment (z, x_u) with tip v in front of w by Rule 7). Also, x_d cannot be behind w (Rule 7 on (x_d, y)). Now, if x_u is behind z, x_u, z and w contradict the monotonicity of the $x_u - x_d$ path. Then x_u must be in a crossover attachment with the tip v behind w, or $v = w$. But then, since x_u does not have priority on v, it would not be matched with x_d. □

We have shown that after running Algorithm 4 on a given matching pair (x_d, x_u), all the unmodified attachments are still valid, and the newly created ones also satisfy the required rules. As before, after processing every single matching pair, we reach the final configuration along with a full attachment scheme. As the handling of the possible *leading-zero*, *down-2up-down* intervals, and unpaired rightmost *down* node is the same as for paths, this completes the proof that a full attachment scheme is maintained in trees. Combining with Lemmas 3 and 4 yields:

THEOREM 5. *Algorithm Tree uses buffers of size at most $O(\log n)$.*

6. CONCLUSIONS

We studied the information gathering problem in paths and trees under the assumption of adversarial traffic. Given an adversary that can inject at most c packets into the network in every step, we showed an $\Omega(\log n)$ lower bound on the buffer space needed to ensure no packet loss. For $c = 1$, we gave deterministic local algorithms that match this bound for directed paths and trees. The existence of local algorithms with $O(\log n)$ buffers for higher rate adversaries remains open. A natural question to ask is if our algorithms generalize to arbitrary routing patterns, or to DAGs. Another intriguing direction for further research is the delay characteristics of our algorithm as well as those of other algorithms proposed in the literature (for example [21]).

7. REFERENCES

[1] A. Miller and B. Patt-Shamir. Buffer size for routing limited-rate adversarial traffic. In *Proceedings of Distributed Computing: 30th International Symposium, DISC 2016*, pages 328–341, 2016.

[2] A. Borodin, J. Kleinberg, P. Raghavan, M. Sudan, and D. P. Williamson. Adversarial queuing theory. *J. ACM*, 48(1):13–38, 2001.

[3] A. Rosén and G. Scalosub. Rate vs. buffer size: Greedy information gathering on the line. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, (SPAA), pages 305–314, 2007.

[4] M. Andrews, B. Awerbuch, A. Fernández, T. Leighton, Z. Liu, and J. Kleinberg. Universal-stability results and performance bounds for greedy contention-resolution protocols. *J. ACM*, 48(1):39–69, January 2001.

[5] M. Andrews. Instability of FIFO in session-oriented networks. *Journal of Algorithms*, 50(2):232–245, 2004.

[6] R. Bhattacharjee, A. Goel, and Z. Lotker. Instability of FIFO at arbitrarily low rates in the adversarial queueing model. *SIAM Journal on Computing*, 34(2):318–332, 2005.

[7] J. Diaz, D. Koukopoulos, S. Nikoletseas, M. Serna, P. Spirakis, and D. Thilikos. Stability and non-stability of the FIFO protocol. In *Proceedings of SPAA*, pages 48–52, 2001.

[8] Z. Lotker, B. Patt-Shamir, and A. Rosén. New stability results for adversarial queueing. *SIAM Journal on Computing*, 33(3):286–303, 2004.

[9] C. Alvarez, M. Blesa, and M. Serna. A characterization of universal stability in the adversarial queueing model. *SIAM Journal on Computing*, 34(1):41–66, 2004.

[10] M. Andrews, A. Fernandez, A. Goel, and L. Zhang. Source routing and scheduling in packet networks. *J. ACM*, 52(4):582–601, 2005.

[11] D. Koukopoulos, M. Mavronicolas, S. Nikoletseas, and P. Spirakis. On the stability of compositions of universally stable, greedy contention-resolution protocols. In *Proceedings of DISC*, pages 88–102, 2002.

[12] E. Anshelevich, D. Kempe, and Kleinberg. J. Stability of load balancing algorithms in dynamic adversarial systems. *SIAM Journal of Computing*, 37(5):1656–1673, 2008.

[13] W. Aiello, R. Ostrovsky, E. Kushilevitz, and A. Rosén. Dynamic routing with fixed size buffers. In *Proceedings of SODA*, pages 771–780, 2003.

[14] W. Aiello, Y. Mansour, S. Rajagopolan, and A. Rosén. Competitive queue policies for differentiated services. *Journal of Algorithms*, 55(2):113–141, 2005.

[15] S. Angelov, S. Khanna, and K. Kunal. The network as a storage device: dynamic routing with bounded buffers. *Algorithmica*, 55:71–94, 2009.

[16] E. Gordon and A. Rosén. Competitive weighted throughput analysis of greedy protocols on DAGs. In *Proceedings of PODC*, pages 227–236, 2005.

[17] Y. Azar and R. Zachut. Packet routing and information gathering in lines, rings, and trees. In *Proceedings of ESA*, pages 484–495, 2005.

[18] G. Even and M. Medina. An $O(\log n)$-competitive online centralized randomized packet-routing algorithm for lines. In *Proceedings of ICALP, Part II*, pages 139–150, 2010.

[19] G. Even and M. Medina. Online packet routing in grids with bounded buffers. In *Proceedings of SPAA*, pages 215–224, 2011.

[20] G. Even and M. Medina. Online packet routing in grids with bounded buffers. *Algorithmica*, 2016.

[21] K. Kothapalli and C. Scheideler. Information gathering in adversarial systems: lines and cycles. In *Proceedings of SPAA*, pages 333–342, 2003.

[22] K. Kothapalli and C. Scheideler. Lower bounds for information gathering in adversarial systems. In *Proceedings of International Conference on Distributed Computing in Sensor Systems*, 2006.

Figures

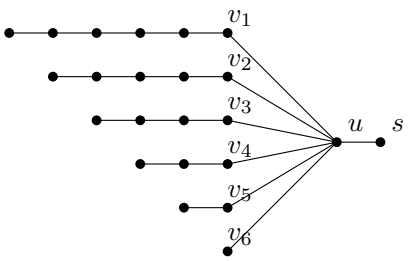

Figure 2: At time step i the adversary inserts c messages into the end of line of length $c-i+1$. As these messages are forwarded toward the sink, at time step c all upstream neighbours of u contain c messages. At this moment, the algorithm selects one of them (w.l.o.g. assume v_1) and forwards its messages to u. Afterwards, the adversary injects c messages, spread almost equally among $v_2, v_3, \ldots, v_c$. Again, the algorithm selects one of $v_2, \ldots, v_c$ (w.l.o.g. assume v_2) and forwards c of its messages to u. Now the adversary spreads c messages among $v_3, v_4, \ldots, v_c$. The process is repeated until only v_c remains. At this moment, the number of messages accumulated at v_c is $\sum_{i=1}^{c} \lfloor c/i \rfloor \in \Omega(\log c)$.

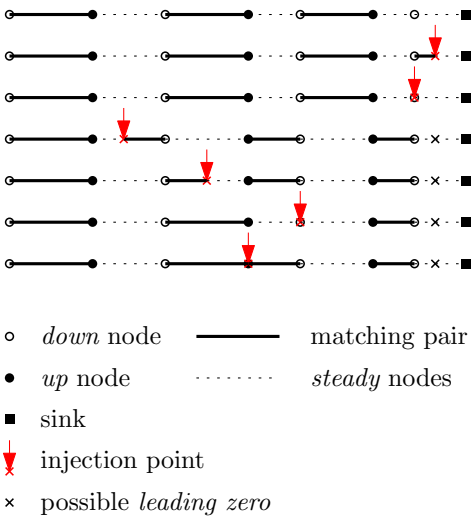

Figure 3: Finding balanced matching for all possible injection cases.

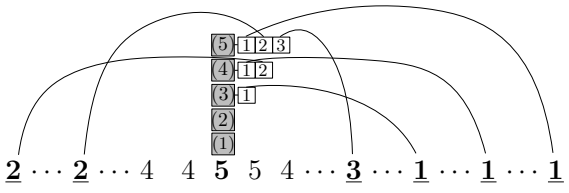

Figure 4: An illustration of a node x of height 5 and all the nodes attached to its packet slots. Only the node heights are shown, and the nodes are depicted from left to right in decreasing order of distance from the sink (i.e. the leftmost node is in the back of all the others, the rightmost in front). The gray boxes correspond to the packets of x, each accompanied with its available slots and attachments. Residues attached to a packet of x appear in boldface and underlined.

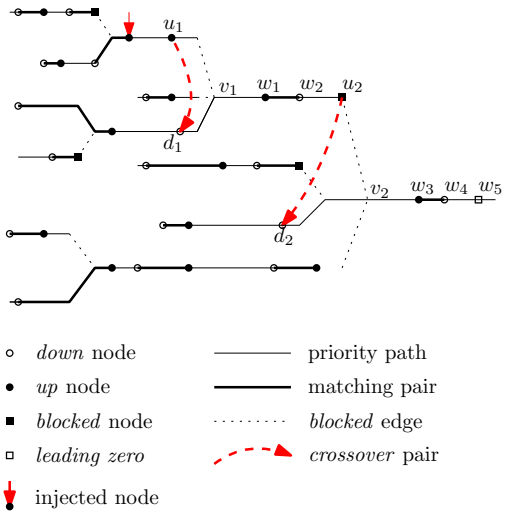

Figure 5: Constructing balanced matching on a tree. Due to adding the (u_1, d_1) matching, the matchings (d_1, w_1) and (w_2, u_2) were removed and replaced by (w_1, w_2), leaving u_2 unpaired. This forced the (d_2, u_2) matching, then switching the (d_2, w_3) and (w_4, w_5) into (w_3, w_4) and leaving w_5 unpaired.

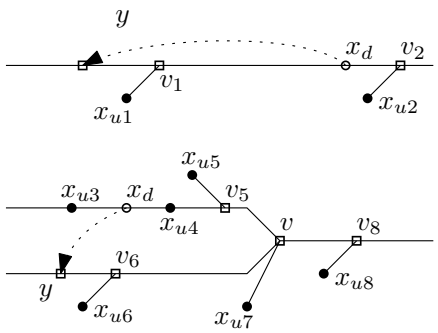

Figure 6: The possible positions of x_u w.r.t. to y and x_d, such that (x_u, y) is a crossover pair.

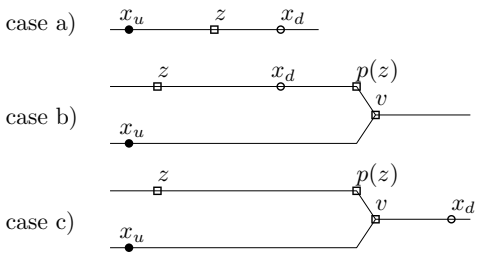

Figure 7: Cases of how z can be behind x_d, while not being behind x_u.

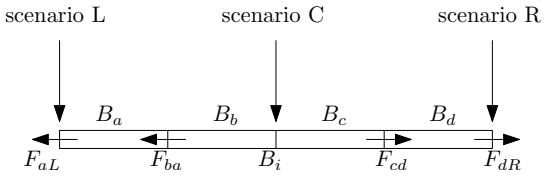

Figure 8: The inductive step: Splitting the block B_i, the relevant injection sites and flows.

Appendix

Proof of Theorem 2. The proof follows the same outline as the proof of Theorem 1, however the induction step is a bit more involved. We reuse the same notation, however some variables are defined differently:

- n_0 is the largest value of form $l4^i$ smaller than n
- $K_i = n_i/4^i$
- $H_i = c(1 + i/4l)$
- $x_i = K_i/4l$

B_i is divided into four equal sized segments B_a, B_b, B_c and B_d (refer to Figure 8)). Note that x_i is set in such a way that x_i time steps are insufficient for the information to cross a quarter of B_i. Denote by M_a, M_b, M_c and M_d the number of messages in those segments, respectively, at time t_i.

The adversary considers three scenarios: Insert c messages at time steps $t_i + 1, \ldots, t_i + x_i$ into

- **Scenario L:** the leftmost node of B_a
- **Scenario R:** the rightmost node of B_d
- **Scenario C:** into the rightmost node of B_b

Let M_x^y denote the number of messages in segment B_x in scenario y at time step $t_{i+1} = t_i + x_i$.

If $M_a^L \geq H_{i+1}K_{i+1}$, the assumptions for stage $i + 1$ are satisfied in B_a at time t_{i+1} and the adversary recurses there. Similarly, if $M_d^R \geq H_{i+1}K_{i+1}$, the adversary recurses in B_d following scenario R. If neither of those cases applies, we have

$$M_a^L < H_{i+1}K_{i+1} = (H_i + \frac{c}{4})\frac{K_i}{4} = \frac{H_iK_i}{4} + \frac{cx_i}{4} \quad (1)$$

$$M_d^R < H_{i+1}K_{i+1} = H_iK_i/4 + cx_i/4 \quad (2)$$

$$M_a^L = M_a + F_{ba} - F_{aL} + cx_i \quad (3)$$

where F_{ba} and F_{aL} are flows of messages from B_b to B_a and from B_a to the left of B_i, respectively, in the time interval between t_i and t_{i+1}, and

$$M_d^R = M_d + F_{cd} - F_{dR} + cx_i \quad (4)$$

where F_{cd} and F_{dR} are flows of messages from B_c to B_d and from B_d to the right of B_i, respectively, in the time interval between t_i and t_{i+1}.

From (3) and (4) we can express

$$F_{ba} = M_a^L - M_a + F_{aL} - cx_i \leq M_a^L - M_a \quad (5)$$

and

$$F_{cd} = M_d^h - M_d + F_{dR} - cx_i \leq M_d^R - M_d \quad (6)$$

where the inequalities in (5) and (6) follow from the fact that $F_{aL}, F_{dR} \leq cx_i$. We are now ready to express

$$\begin{aligned} M_b^C + M_c^C &= M_b - F_{ba} + M_c - F_{cd} + cx_i \geq \\ &\geq M_b - M_a^L + M_a + M_c - M_d^R + M_d + cx_i \end{aligned} \quad (7)$$

Applying the induction hypothesis and substituting (1) and (2) into (7) we obtain

$$M_b^C + M_c^C \geq H_iK_i - M_a^L - M_d^R + cx_i \geq \frac{H_iK_i}{2} + \frac{cx_i}{2} \quad (8)$$

hence, at least one of M_b^C and M_c^C is at least $H_iK_i/4 + cx_i/4 = H_{i+1}K_{i+1}$ and the adversary can recurse there.

This argument holds as long as $x_i \geq 1$, i.e. $K_i \geq 4l$. The number of stages is then $\log_4(n_0/4l) = \lfloor \log_4(n/l^2) \rfloor - 1$, resulting in maximal buffer size of at least

$$c(1 + (\log n - 2\log l - 2)/8l) \in \Omega(c \log n/l)$$

□

Note: As for the directional case, the proof can be straightfowardly modified for burstiness δ.

Proof of Theorem 3. *Upper bound (sketch):* Consider how a node v with maximum buffer size k can increase its buffer size. Since it has maximum buffer size, it must forward a message to $s(v)$ according to the algorithm *Downhill-or-Flat*. Hence, the only way to increase its buffer size is to have a message injected into it by the adversary, and simultaneously receive a message from its predecessor u. Hence, u must be of height k as well. How could u have reached height k? While it was at height below k, it was not sending a message to v, hence for v to simply maintain its height, it must have been receiving injections from the adversary. Therefore, u could not have been receiving injections and the only way to reach height k is if it and its predecessor u' were at height $k - 1$. Now, for u' to reach height $k - 1$, it must have had a predecessor u'' of height $k - 2$. Summing up, for v to reach height k, there must have been $sum_{i=1}^{k} k = k(k + 1)/2$ messages in the system to the left of v. However, unlike *Downhill*, *Downhill-or-Flat* ensures that no "holes" of empty-buffer nodes are created to the right of non-empty nodes and messages efficiently travel to the sink. Hence, the total number of messages in the system cannot exceed n, i.e. $k(k + 1)/2 \leq n$ and therefore $k \in O(\sqrt{n})$.

Lower bound: Let j be an integer. Consider a directed path of $n = j(j + 1)/2 + 1$ nodes, with all buffers being initially empty. The adversary injects works in two stages. In Stage 1, by repeatedly inserting a packet in the leftmost

empty buffer $n-1$ times, according to the *Downhill-or-Flat* algorithm, the packets are forwarded to the right. Thus, at the end of Stage 1 we obtain one buffer of size 0 followed by a sequence of $n-1$ buffers of size 1, and the sum of buffer sizes is $n-1$.

In stage 2, the adversary injects $j(j-1)+1$ times into the rightmost buffer of height 1. Since initially the buffer sizes are the same (except for the leftmost buffer), due to the use of the *Downhill-or-flat* algorithm, the following relationships can be observed in Stage 2:

1. The sum of all buffer sizes remains $n-1$.
2. If $s(i) \geq s(i+1)$ for some i then a packet is sent from node i to node $i+1$, thus increasing the number of packets in the last $n-i-1$ buffers.
3. $s(i+1) < s(i)+2$.

Thus, there is a flow of packets in the buffers to the right, but this cannot continue forever due to items 1 and 3 above. Thus eventually a configuration is reached in which $si < s(i+1)$.

It can be observed that for $n = j(j+1)/2+1$ this occurs after $j(j-1)+1$ insertions in Stage 2, when the sizes of the j rightmost buffers, listed from the left, are $1, 2, \ldots, j$. In this case $j = (\sqrt{1+8n}-1)/2$. See the example below for $j = 4$, i.e., $n = 11$. □

Beginning of Stage 1	Beginning of Stage 2
01000000000	01111111111**1**
01100000000	00111111112**1**
01110000000	00011111212**2**
01111000000	00001112122**2**
01111100000	00000121222**2**
01111110000	00000112132**2**
01111111000	00000021223**3**
01111111100	00000012133**3**
01111111110	00000011233**3**
01111111111	00000002233**3**
	00000001333**3**
	00000001243**3**
	00000001234

Figure 9: Example run of Algorithm *Downhill-or-Flat* for $j = 4$, $n = 11$.

Proof of Lemma 7. We prove each fact separately:

1. If a residue becomes an *up* node, its attachment is removed on line 13.
2. Note that since residues do not go down, x_d has no slot it needs to be detached from (observe that x_d can only become a residue after being processed, and hence when prcessing (x_d, x_u), x_d could not have been a residue). Thus an existing slot can become empty only by detaching the residue x_u on line 13. Line 5 makes sure that if x_d was the guardian of x_u, the corresponding spot will disappear and hence not become empty. If $z \neq x_d$ was the guardian of x_u, lines 15 and 18 ensure that the slot that was connected to x_u won't become empty in case of $h_d > h_u$, while Lemma 5 ensures that $h_d = h_u$ is impossible.
3. The only new slots are the $h_u - 1$ slots of x_u acquired by raising its height to $h_u + 1$. Passing of the attachments from x_d to x_u on line 7 ensures that slots $x[h_u+1, i]$ for $i \leq \min(h_d - 2, h_u - 1)$ are filled by attachments transferred from x_d. As $h_d \geq h_u$, this leaves only the slot at height $h_u - 1$ potentially unfilled, which is then filled by attaching x_d to it at line 9.

4a Note that only residues of height at most $\min(h_d - 2, h_u - 1)$ are transferred. The statement then follows directly from the second part of Lemma 2.

4b The interval between z and x_d consists of interval from z to x_u, where the statement follows from Rule 5, and of the interval from x_u to x_d, where it follows from Lemma 2.

4c The statement follows from Rule 5 applied between z and x_u, and between x_d and y, combined with Lemma 4 for the nodes between x_d and x_u.

5. Follows directly by combining Fact 4 with Rules 3, 4 and 5 of the attachment scheme.

□

Proof of Lemma 9. For the purpose of the proof, we refine the notion of priority for an intersection node v that does not receive a packet from any of its predecessors: if a predecessor v' of v is in front of the injected node t, then assign v' as the priority node; otherwise, assign priority arbitrarily among the predecessors of v. Note that this priority is purely conceptual and does not alter the behavior of the packets being sent in the current round, though it does affect the construction of priority lines (which is fine since priority lines exist solely for the purpose of the analysis).

As discussed, each *up* node of the tree can be matched with a *down* node on the same priority path, with the possible exception of one *up* node x_u, which is the last non-steady node of the injected path p. Let v be the first intersection in front of x_u for which p does not have priority, and let p_v be the priority line containing v. Then x_u is matched with x_d, the first *down* node on the p_v path behind v. We claim that (x_u, x_d) is a valid pair, i.e. that there is no *up* node between x_d and v (except possibly v). To see this, it suffices to observe that v must receive from its predecessor v' on the p_v path (otherwise, v receives from none of its predecessors, and by our notion of priority, p would contain v since x_u is on the injected line). The first non-steady node v'' behind v on the p_v path is easily seen to be a *down* node, since each node on the $v'' - v$ path must send a packet forward (except perhaps v). Thus $(x_u, v'' = x_d)$ is a valid pairing.

Now, there may have been an *up* node w on p_v that was previously matched with v''. In this case, we must partition the non-steady nodes in front of v'' into *up-down* pairs (which is possible since p_v does not contain the injected node). If p_v is not the drain, this leaves the last non-steady node w_u of p_v unmatched. Because w is in front of the injected node, we can apply the same arguments as above to show that w_u will be matched in a valid crossover pair with some *down* node w_d. One can see that by repeating this process repeatedly, we will always form crossover pairs (z_u, z_d) such that z_u is in front of the injected node and z_d is not. Moreover, we will eventually reach such a pair in which z_d lies on the drain, in which case the partition into *up-down* pairs does not leave an unmatched *up* node. □